A growing number of ham shack projects involve the use of microcontrollers, or MCUs, and that often involves writing computer code to tell the MCU what to do and how to do it. Since many of us are more familiar with Morse code than computer code, KZ1G offers a tutorial on the basics of writing these instructions.

An Introduction to Microcontrollers

BY JONATHAN TITUS,* KZ1G

Hams and electronic experimenters like microcontrollers because they provide ways to create devices that would otherwise need dozens or hundreds of discrete components. Instead of making you rely solely on hardwired connections, a microcontroller (MCU) provides a way to create a basic design and then program the functions needed. And MCUs already include internal modules for serial communications, event timing, motion control, USB communications, and digital inputs and outputs, among others. Some MCUs include analog-to-digital and digital-to-analog converters, and industrial-network functions. MCUs do the same things as mainframe computers: Input and output information, store programs and data, manipulate data, and make decisions. Only the scale and complexity differ.

When you want to design a controller for an antenna rotor, for example, an MCU and some high-current transistors along with a control program you create can do the job. Want to add tones[1] at the start and end of VHF communications? An 8-pin MCU can handle the task for under $1. You program the MCU's internal oscillators and the two tones appear on an output pin.

You can buy MCUs from semiconductor manufacturers such as Texas Instruments, Freescale Semiconductor, Parallax, and Microchip Technology. Choices center on the requirements for an application, MCU cost, ease of use, availability of programming tools, and cost of programming and testing hardware. I like to start with an evaluation board or a development kit that provides an MCU IC and additional components needed to get started. This type of all-in-one board should include a USB connection for communication with a host PC on which you'll create and download programs. It helps to have a few LEDs and pushbuttons so you don't need to add anything else to get started.

Give the Propeller a Spin

You will learn about the Parallax Propeller MCU in this article because it gives us an easy way to learn how to use an MCU. Parallax sells a Propeller QuickStart P8X32A board (p/n 40000) for $35 and I recommend it highly (*Photo A*). If you buy a QuickStart board, spend another $1.50 on the QuickStart Pin Finder (p/n 721-00017) to help you quickly identify board signals for external devices and sensors[2].

Why not start with an Arduino, RaspberryPi, or BeagleBone? The basic Arduino board runs on 5 volts. Many new ICs require a 3.3-volt supply, so projects might require logic-level conversion. That means extra components. The RaspberryPi and BeagleBone rely on Linux, which can tie a beginner in knots for days or weeks.

You need only a USB Type-A to Type-Mini-B cable and a PC to use a QuickStart board, along with some free software, which I'll explain shortly.

Programmers and experimenters use either the SPIN or C language (or both) to write code for the Propeller MCU. SPIN offers the easier route for people with little or no program-

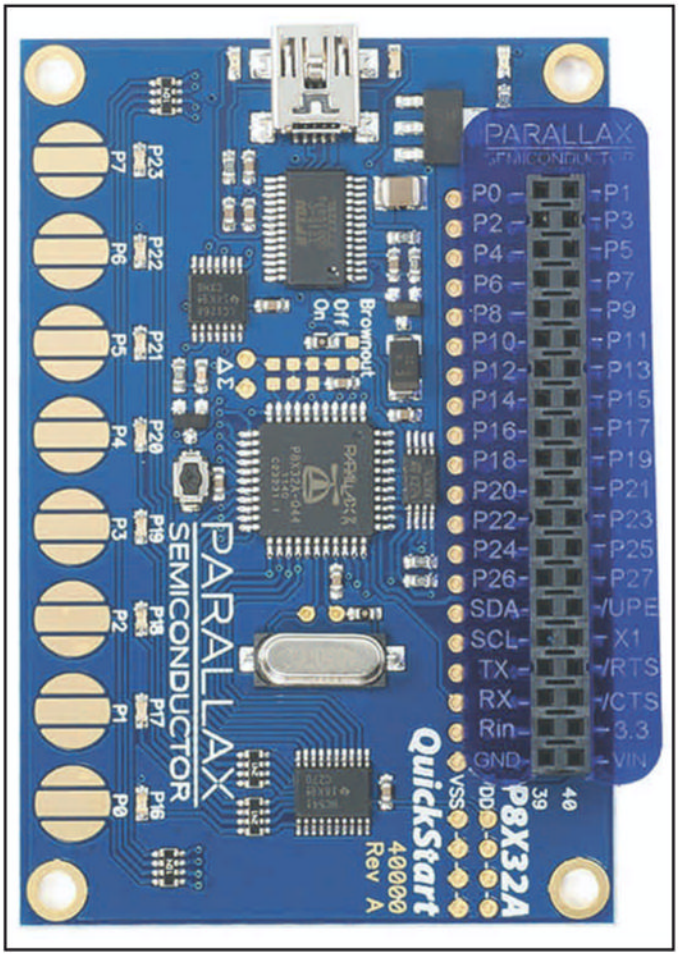


Photo A. A Parallax QuickStart board with a blue plastic Pin Finder slipped onto the 40-pin I/O expansion connector.

* *14253 S. Trailview Way*
Herriman, UT 84096-8029
<jontitus@comcast.net>

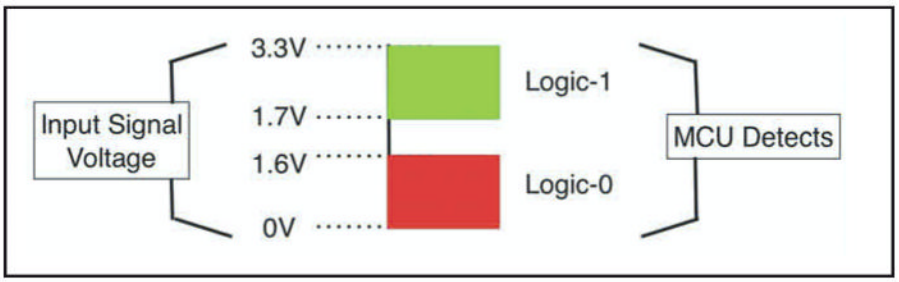


Figure 1. Voltage levels associated with a logic-0 and or a logic-1 signal for 3.3-volt ICs.

ming experience, so we'll start there. If you learn to write good code and learn programming concepts in one language, you'll easily write good code in C, BASIC, Java, and other languages.

Control I/O Pins and LEDs

Before you program a QuickStart board, you need to know what the board provides. The MCU has 32 input/output (I/O) pins, four of which get dedicated to USB communications with your PC, and to on-board control. The remaining 28 pins get divided between touch switches, LEDs, and uncommitted input/output (I/O) pins. These on-board devices have names that let you quickly identify them in programs:

- Eight blue LEDs at pins P23 through P16
- Touch switches at pins P7 through PØ
- Uncommitted I/O at pins P15 through P8 and at pins P27 through P24.

Each I/O pin has a 1-bit *direction memory* associated with it. A logic-Ø in this bit sets the corresponding pin as an input for logic signals. A logic-1 in this bit sets the pin to act as a logic output. The Propeller I/O pins work with 3.3-volt logic levels (*see Figure 1*). We consider the eight LEDs as output devices because the MCU sends a signals out to turn an LED on or off.

The SPIN instruction **DIRA[23..16] := %11111111** causes the MCU to set the *direction* of electrical signals at pins P23 through P16s as outputs. In this example, the **[23..16]** information identifies a range of pins we want to control. The 8-bit value **%11111111** provides a logic-1 value for the direction memory of each I/O pin. The percent symbol (%) prefix identifies a binary (base 2) value. And we call individual logic-1s and logic-0s bits.

You could write the same DIRA command with the equivalent decimal value, **DIRA[23..16] := 255**, or with the equivalent hexadecimal value **DIRA [23..16] := $FF**. I find binary patterns the easiest to identify pin directions; I can see exactly what happens at each pin. The Propeller MCU can work with binary, decimal, or hexadecimal values. They all eventually become groups of 1s and Øs to the MCU.

You may use the DIRA command to set an individual pin or pins as inputs or outputs. The code **DIRA[23..20] := %1111** sets four pins as outputs to control only four LEDs. Likewise, the command **DIRA[12] := %0** sets an uncommitted pin as an input, perhaps for a signal from a sensor.

In the Spin language, the **:=** operator indicates an assignment of the *value* on the right to the *variable* on the left. Assignments such as this always go from right to left. The following line shows the assignment of decimal 153 to the variable Alpha:

Alpha := 153

Now a program can use Alpha to represent 153:

Beta := Alpha + 2

As a result, Beta equals 155.

Now back to the eight LEDs. Suppose you want to display an alternating pattern of on and off LEDs; off, on, off, on, and so on. On the QuickStart board, a logic-1 signal turns on the corresponding LED. A logic-Ø turns it off. Thus this command:

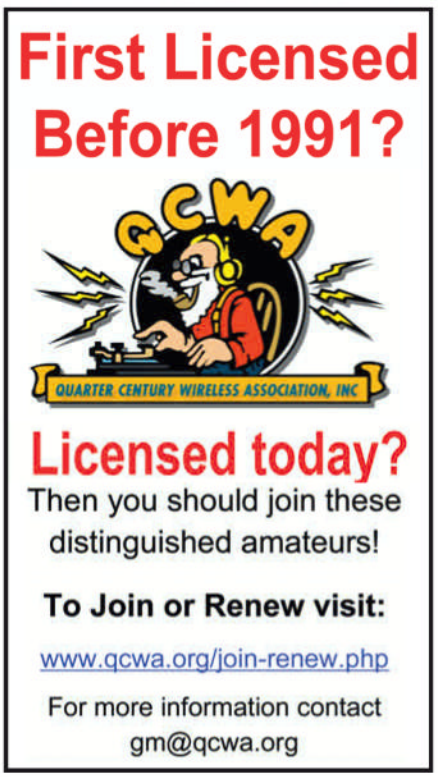
OUTA[23..16] := %01010101

would turn off LEDs P23, P21, P19, and P17. The LEDs at P22, P20, P18, and P16 would turn on.

Install the Tools

Before you use the DIRA and OUTA commands in a simple program, you must understand how to use the Propeller Tool software and how to find reference information. It takes only a few steps and a little practice to make you a whiz at LED control. To start, download the free Propeller Tool software from the Parallax site:<https://www.parallax.com/downloads>. You will also need the USB drivers. The Propeller Tool Software page shows instructions for software setup. You may use the Tool even without a Propeller IC or board. If you have a QuickStart board, *DO NOT* connect it to your host PC until you have installed the Tool software.

After you install the Propeller Tool, create a subfolder within your Documents directory and label it "Propeller Workspace." Your Propeller programs and information should get saved here. Next, start the Propeller Tool program and connect your QuickStart board. A small green LED turns on to indicate the board has power via the USB cable. In the upper menu line, click on Run and then click on Identify Hardware ... The PC should display a message that lets you know the Propeller Tool had a QuickStart board connected properly (*Photo B*). Your COM-port value will likely differ from mine. If you see an error message (*Photo C*), disconnect your hardware and close the Propeller Tool software. Restart the Propeller Tool and, when you see the Prop Tool screen, plug in the QuickStart board and try again. If



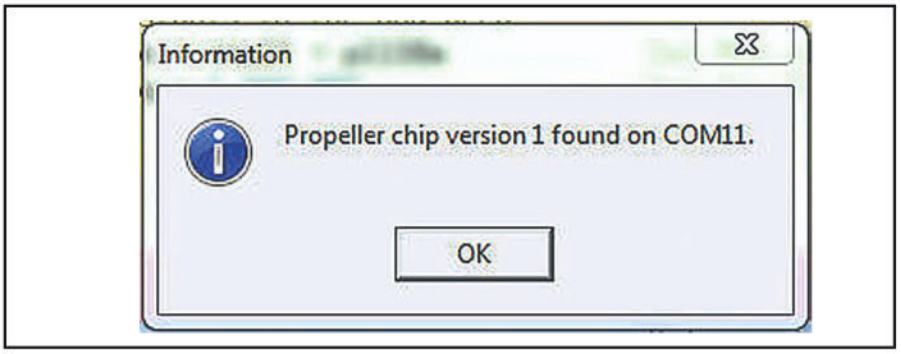


Photo B. This message appears when the Propeller Tool detects a Propeller MCU.

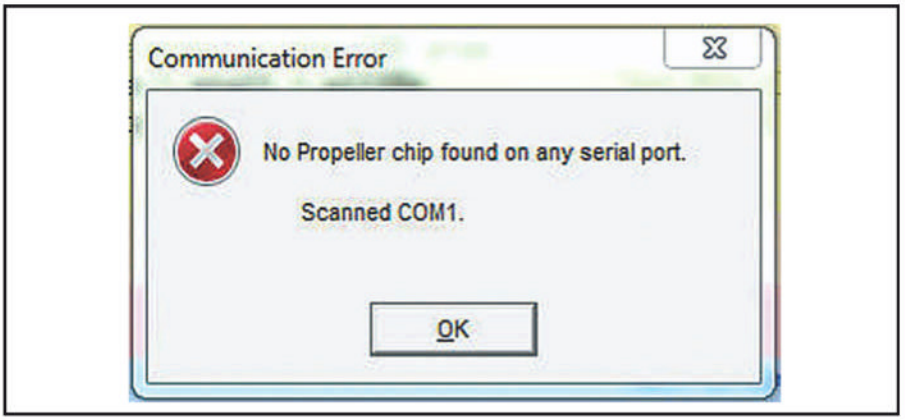


Photo C. This error message indicates either a disconnected QuickStart board or lack of the proper USB drivers.

you still have problems, you might need to load a virtual-serial-port USB driver available from Parallax.

Run an LED Program

The Propeller Tool includes a template you can use to create a program. In the upper menu line, select File and then New (From Template), which opens a "blank" program ready for your instructions. The color-coded sections identify places for specific information such as constants (CON), variables (VAR), and public (PUB) code. The code you want to try goes in the PUB area. The CON section includes two lines that each program MUST supply:

```
CON _clkmode = xtal1 + pll16x
    _xinfreq    =  5_000_000
```

These commands let the MCU know the external crystal's operating frequency and set up the on-chip oscillator. Without this information, a program cannot run. (The underscores in 5_ØØØ_ØØØ do nothing but separate values to make them easier to read. The SPIN software ignores them.)

Within the PUB section, type the following information. Pay careful attention to the indentation of commands (more on indentations later):

```
PUB  Start                          'Main program starts here
    DIRA[23..16] := %11111111  'Set all 8 LED pins as outputs
    OUTA[23..16] := %01010101  'Send this pattern of bits to
                                        ' the LEDs. 0 = off, 1 = on
    repeat                          'Program ends here, MCU sits in
                                        ' a dummy loop that does nothing
                                        ' but waste time.
```

The text after the apostrophe has no effect on the program. These *comments* simply help document our programs so you and others know what should happen. Good programmers also include a *comment block* at the start of a program to give the code a title, a date, a version number, the programmer's name, and

From MILLIWATTS To KILOWATTS SM
In Stock Now!
Semiconductors for Manufacturing and Servicing Communications Equipment
Visit Our Website
• RF Modules
• Semiconductors
• Transmitter Tubes
Se Habla Español • We Export
Phone: 760-744-0700
Toll-Free: 800-737-2787
(Orders only) 800-RF PARTS
Website: www.rfparts.com
Fax: 760-744-1943
888-744-1943
Email: rfp@rfparts.com
RF PARTS COMPANY

Collins Radio
Repair & Tune-Up DVD Guides

Restoring or repairing classic Collins S-Line equipment? These DVDs are like having an experienced professional right next to you!

From Hi-Res Communications, Inc., these well-produced, authoritative DVDs cover all the most common repair and tune-up subjects on these classic radios.

Collins KWM-2
Two disc set, 236 minutes total
Order No. C-KWM **$89.95**

Collins 75S-3/32S-3
Two disc set, 226 minutes total
Order No. C-75S **$89.95**

Collins 30L-1
Single Disc 61 minutes total
Order No. C-30L **$39.95**

Shipping & Handling: US -add $7 for the first item, $3.50 for the second and $2 for each additional item. CN/Foreign: $25 for the first item, $10 for the second item and $5 for each additional item

CQ Communications, Inc.
17 West John St. Hicksville, NY 11801
http://store.cq-amateur-radio.com
800-853-9797
FAX us at 516 681-2926

a description of what the program will do. If the program relies on any other code, this information should appear here, too.

Before you can run the LED program, remove all indented information in the VAR, OBJ, PRI and DAT sections. You may remove the PRI and DAT headings, too. If you leave this information in place, the software tool will treat it as part of your program, which it isn't, and you'll see error messages.

To run your program, go to the upper menu line, select Run, Compile Current, and then Load RAM. (On some PCs, the F10 key has the same function.) Now you should see the P22, P20, P18, and P16 LEDs turn on. You can experiment with different LED patterns. You've successfully created a program that caused a Propeller MCU to control eight external devices — LEDs.

When you rerun the program, the four LEDs turn off briefly and then turn on again. Not much to see. You'll use the LED program again, so go to the File menu and select Save as ... give your program a name such as LED_Control_1.spin and save it in the Propeller Workspace folder.

Although the QuickStart board has controlled only LEDs so far, the outputs could control logic circuits, a digital-to-analog converter, communications with a transceiver, motors, signs, valves, an antenna tuner, and so on.

Add Spin Objects

When programmers can re-use software they save time and work with known-good code already tested by others. That's one fewer thing that can go wrong. You may not steal code, but you may use sections of code called *objects, functions, procedures*, and *subroutines* created for specific purposes and considered *open source*. The Propeller Object Exchange (OBEX) simplifies programming because people freely contribute SPIN objects to help others. Large software "packages" of Propeller objects might contain a dozen or more public (PUB) objects that cause time delays, transmit information via serial communications, control sensors, and so on.

The set of objects in the Timing.spin program lets us add delay commands to the LED program so we can see the patterns change. Download the Timing.spin file from <http://obex.parallax.com/object/820>. You also can browse or search for code for a specific task. Save the Timing.spin software in your Propeller Workspace folder. Then go to the Propeller Tool and use the "file finder" on the left side of the screen to locate your workspace folder. Open it and look for the Timing.spin name in the bottom file-name window. Click on Timing.spin and you should see a second tab at the top of the large color-coded editing window.

Click on the Timing.spin tab so you can examine the code. You will see five objects that start with the PUB prefix. The public object pause1s(period) lets a program create delays in 1-second increments. You don't alter the object, though. Instead, the LED program *links* to the Timing.spin objects, and the LED code refers to timing objects.

Here's how the process works. In the LED program's red OBJ section, the LED program must identify the objects in the Timing.spin software with a prefix that means something to you, say, MyDelays. So the OBJ section would look like this:

```
OBJ
        MyDelays  :  "Timing"                  'Use objects in Timing.spin
```

Now the LED program can use the objects in Timing.spin. The code gives these objects the MyDelays. prefix. The MyDelays name requires a period (.) between it and the object name. So to create a 2-second delay in the LED program we would add the statement:

```
MyDelays.pause1s(2)
```

In essence, this statement says, use the object pause1s(period) in the program I refer to now as MyDelays. And send the pause1s(period) object the value 2 for a 2-second delay. After the delay, control returns to the LED program. We don't care how the Timing.spin object works; it just does.

LEDs and Timing Objects

Given a useful time-delay object, you can create a loop in which the LED pattern can change every two seconds. Before you change the LED program, save it again, but give it a new version number, such as: LED_Control_2.spin. Now you have the older version to fall back on if you have a problem with version 2.

Switch to the LED version-2 program and make the following changes:

1. In the OBJ section, add:

```
OBJ
        MyDelays  :  "Timing"                 'Use objects in Timing.spin
```

2. Delete the **repeat** command at the end of the LED program.

3. Add the following statements to your LED program:

```
PUB  Start                                          'Program starts here
   DIRA[23..16] := %11111111  'Set LED pins as outputs
   repeat                                           'Begin repeat loop
     OUTA[23..16] := %01010101          '1st LED pattern
         MyDelays.pause1s(2)                  'Wait 2 seconds
     OUTA[23..16] := %10101010          '2nd LED pattern
  MyDelays.pause1s(2)                   'Wait 2 seconds
```

Check your program for accuracy and then run it. The LED patterns should shift every two seconds. Ensure you have the LED program saved in the Propeller Workspace folder along with the LED_Control_2.spin program.

In the PUB section shown above, the commands under the repeat statement comprise the operations that will repeat. The *indentation* identifies the repeated commands. If you do not properly indent commands, you'll see unexpected results. Reference information for each SPIN command explains the necessary format, if any.

Try a Binary Counter

To see more action on the LEDs, create a counter that displays a new binary value every second. To do so the program needs a variable for the increasing count. In the orange variables (VAR) section, add:

```
VAR
byte    LEDCounter                      'Variable for LED-count value
```

The keyword *byte* indicates we want an 8-bit storage location set aside for the counter value, named LEDCounter. Now the program section under PUB needs a few changes. The listing below shows the code needed:

```
PUB  Start                                          'Start program here
   DIRA[23..16] := %11111111          'Set LED pins as outputs
   repeat                                           'Begin repeat loop
     OUTA[23..16] := LEDCounter                 'Display counter value
         MyDelays.pause1s(1)                    'A 2-second delay
     LEDCounter := LEDCounter + 1       'add 1 to the count
                                                    'Loop ends here
```

The OBJ section of the LED program should still look like this:

```
OBJ
        MyDelays  :  "Timing"                 'Use objects in Timing.spin
```

The SPIN language provides for three types of variables, *bytes*, *words*, and *longs*. These variables can store 8, 16, or 32 bits respectively. Before you use a variable in a program, ensure it has a proper definition in the VAR section.

The SPIN language comprises 80 commands, but most programs use only a few of them. I recommend SPIN programmers download the latest version of the "Propeller Manual," available at: <http://bit.ly/1LFJvkX>. Save this 400-page document in the Propeller Workspace folder for quick reference to commands.

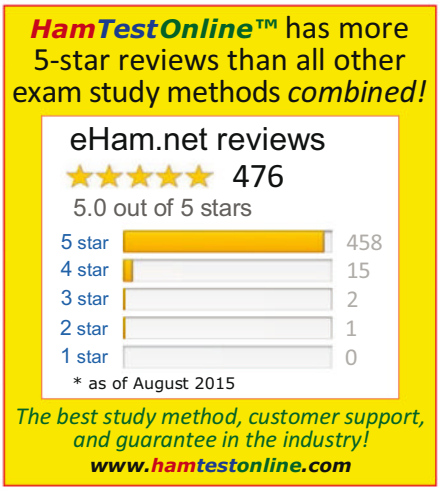
Conclusion

I hope this has helped unravel any mystery around writing instructions for microcontrollers. I encourage you to give it a try. Once you've mastered these basics, you can move on to building controllers that can help meet your specific needs at your station.

Notes:

1. Quindar tones in radio communications came into use at NASA during the Apollo space missions.
2. The author has no commercial interest in the Propeller MCU or Parallax products.

THE WIREMAN, INC™
"CERTIFIED QUALITY"
800-727-WIRE (9473)
Still, going strong after 35 years! The "Keywords" for
"Certified Quality" Wire, Cable, Connectors, Accessories,and
customer service. See it all at www.thewireman.com
Tech Help: 864-895-4195 or info@thewireman.com
SOUTHWEST US? Call 405-376-9473
TOP WIREMAN dealer CLEAR SIGNAL PRODUCTS
www.coaxman.com • orders@thecoaxman.com



KZ1G continues his tutorial on the basics of programming and using the microcontrollers (MCUs) that are becoming ever-more-common in ham radio projects.

An Introduction to Microcontrollers, Part II

BY JONATHAN TITUS,* KZ1G

The previous microcontroller article[1] explained how software can control digital outputs that turn LEDs — and other devices — on or off. The **DIRA** command set MCU pins as either inputs or outputs, and the **OUTA** command transferred a logic-1 or a logic-0 to identified output pins. You also learned about objects that perform operations independent of your code, and you used an object in the Timing.spin code to create a 1-second delay (see *Table 1*).

That program also defined a byte (8-bit) variable, *LEDCounter*, that has 1 added to it each time through the repeat loop. Thus the value of *LEDCounter* can range from 00000000_2 to 11111111_2, or 0 to 255. You might have wondered what happens when a pass through the loop adds 1 to 255. The *LEDCounter* value becomes 0. Variables do not include a "carry" or an "overflow" bit, so although we know 255 + 1 = 256 ($1\underline{00000000}_2$), the 8-bit *LEDCounter* variable will hold only the eight *least*-significant bits — those eight on the right side of the value (underlined).

In this program, the MCU repeats the same instructions again and again (*Figure 1*). An *unconditional* repeat statement controls the loop, so the MCU has no way "break out" of, or exit, the loop. For that reason, unconditional loops get used infrequently.

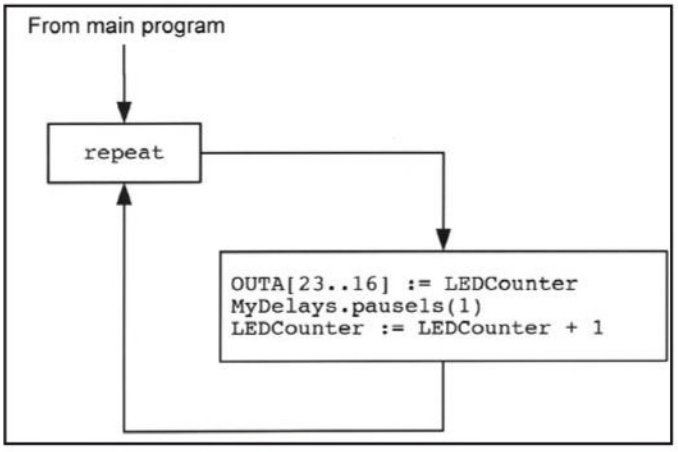


Figure 1. An unconditional repeat loop gives an MCU no way to "break out" and execute other instructions. Thus the MCU ends up in a never-ending, or infinite, loop.

Practical Programs Make Decisions

Unlike the simplistic LED-counting program, practical programs include instructions that let an MCU decide whether to proceed with one set of instructions, or an alternate path of instructions.

In the case of Spin-language repeat loops, we include a statement that tests for a specific condition. The result lets the MCU determine whether to run through the repeat loop again, or exit from it. The process involves a *condition test* that can have only True or False as the answer. The Spin language defines True and False as "built in" constants. Now you'll learn how to program the Propeller MCU to make decisions based on conditional-repeat statements. (*Editor's note: The Propeller microcontroller was introduced in Part I and is used for the examples in this series. See October's CQ for more.)*

The first type of condition test in a repeat loop uses an "until" statement, and the second type uses a "while" statement. Suppose you want an 8-bit binary-counter loop to stop when the *LEDCounter* value *exceeds* 27 (00011011_2). You would use the repeat statement shown here:

```
repeat until LEDCounter > 27
```

The test, **LEDCounter > 27**, determines whether or not the *LEDCounter* value exceeds 27. (The > symbol indicates a "greater-than" test.) When *LEDCounter* has values from 0 to 27, the test produces a False condition because those values do not exceed 27. That result lets the loop repeat again.

When the *LEDCounter* value becomes 28, the test determines *LEDCounter* now exceeds 27, so the test yields True. This condition causes the *repeat until...* loop to end and the MCU moves past the loop code to instructions that follow (*Figure 2*). To ensure proper operation of instructions below, but not included in the loop, always outdent them to align with the letter r in the *repeat until...* statement. Then only the indented statements and commands become part of the loop.

To test this type of conditional-repeat statement on the Propeller MCU, start the Propeller Tool program on your PC. Then open the *LED-counter_V2.spin* program you wrote, tested, and saved while reading the previous article. If you don't have that program, find it in the Propeller Object Exchange (OBEX): <ttp://obex.parallax.com.object/821> The downloadable ZIP file includes the *Timing.spin* program.

The code snippet in *Table 2* highlights the changes in bold. In addition to changing the first repeat statement, add an

** 14253 S. Trailview Way*
Herriman, UT 84096-8029
<jontitus@comcast.net>

unconditional repeat command at the end. Then orient the QuickStart board so you have the small USB connector on the left. When you run the modified program, what do you see on the LEDs after about 30 seconds? You should see the LED pattern 00011011, where 1 represents on and 0 represents off. That binary value equals 27. In this example the loop continued for as long as the condition test gave a False result. A True result forced the MCU out of the loop. The MCU then executed the stand-alone unconditional repeat com-

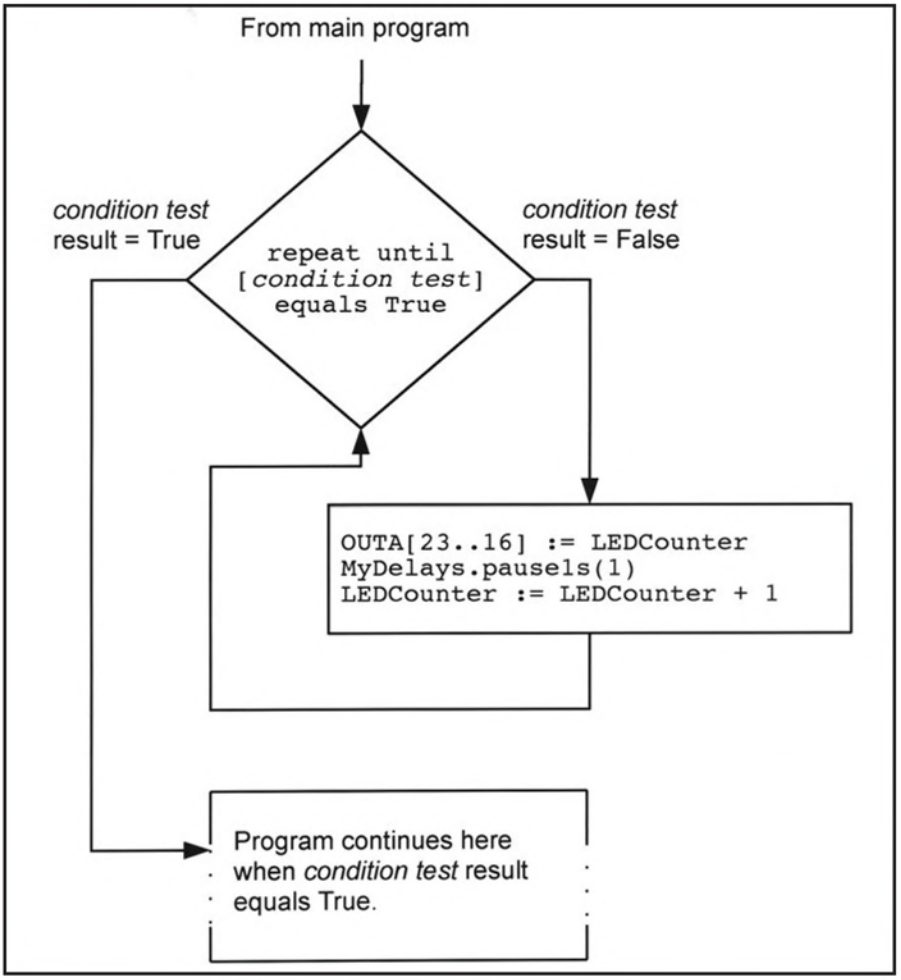


Figure 2. A repeat-until loop will run until the result of the condition test becomes True.

```
CON
    _clkmode = xtal1 + pll16x          'Crystal 1 and PLL 16x
    _xinfreq = 5_000_000               'Crystal freq 5 MHz

VAR
byte    LEDCounter                     'Variable for LED-count value

OBJ
        MyDelays  :  "Timing"          'Use objects in Timing.spin

PUB  Start                             'Start program here
    DIRA[23..16] := %11111111          'Set LED pins as outputs
    repeat                             'Begin repeat loop
      OUTA[23..16] := LEDCounter       'Display counter value
          MyDelays.pause1s(1)          'A 1-second delay
      LEDCounter := LEDCounter + 1     'add 1 to the count
                                       'Loop ends here
```

Table 1. The basic one-second delay program, plus LED counter routine, written in Part I of this series, and modified here in Part II.

From MILLIWATTS To KILOWATTS℠

In Stock Now!

Semiconductors for Manufacturing and Servicing Communications Equipment

Visit Our Website

• RF Modules
• Semiconductors
• Transmitter Tubes

Se Habla Español • We Export

Phone: 760-744-0700
Toll-Free: 800-737-2787
(Orders only) 800-RF PARTS
Website: www.rfparts.com
Fax: 760-744-1943
888-744-1943
Email: rfp@rfparts.com

VISA MasterCard AMERICAN EXPRESS Cards

RF PARTS COMPANY

mand, which simply gave the MCU something to do endlessly.

If you remove the repeat command at the end of the program, what happens when you run the program? The MCU resets and turns off all LEDs. The "do nothing" repeat command simply stops the MCU and does not let it reset the LED pins.

Repeat While We Run a Test

The second type of condition test uses a "while" statement to determine when the MCU will terminate a loop. The line below shows the syntax:

repeat while LEDCounter < 13

The flow chart in *Figure 3* shows this type of repeat-while operation. As long as the *LEDCounter* variable has a value *less than* 13, the test LEDCounter < 13 produces a True condition, and the loop continues.

Change the *repeat until...* statement in your program to the *repeat while...* statement shown above. Run the revised program and watch the LEDs. What result did you see? Did you expect that value?

The LEDs now show the pattern, 00001100, which equals 12. Why didn't the MCU display 13 (00001101) on the LEDs? When *LEDCounter* becomes 13, the condition test LEDCounter < 13 yields False because the value 13 in *LEDCounter* is not less than the 13 in the statement. Thus the count on the LEDs stops when program displays 12 on the LEDs.

So far, the *repeat-until* and *repeat-while* statements have used the greater-than (>) or less-than (<) operators to determine when to end a loop. The Spin language — and others — provide additional operations you can use in True-or-False tests:

Use <> for not equal to (a less-than symbol followed by a greater-than symbol)
Use == for equal to (two equal signs)
Use =< for less-than OR equal-to
Use => for greater-than OR equal to

If you change the *repeat while...* condition test to:

repeat while LEDCounter =< 13

the loop will display a final value of 13, or 00001101_2. The test yields a True result for values less than OR equal to 13.

Keep in mind when you use a *repeat while...* or a *repeat until...* loop, you must update the condition-test variable *within* the loop. The example in *Table 3* illustrates the problem if you do otherwise. The LEDCounter gets set to 0 to just before the repeat loop starts. Then the loop runs... and runs … and runs forever because nothing within the loop changes the *LEDCounter* value. It remains 0.

Likewise, pay careful attention to indentation, which determines which instructions run as part of a loop. The code snippet in *Table 4* illustrates an indentation problem. This loop also runs forever and all LEDs remain off (00000000) because the statement to add 1 to the LEDCounter variable will run only *after* the loop ends, which this loop never does. Because the programmer forgot to indent the LEDCounter := LEDCounter + 1 statement, it does not run within the repeat loop. Indent it and the loop operates properly.

In the Propeller Tool editing window, look carefully at the light-gray lines to the left of your code. These lines identify the instructions that run within a loop or within other sections of code. Unfortunately, the lines do not appear when you print a code listing from the Propeller Tool, and they're a challenge to see even on a good display.

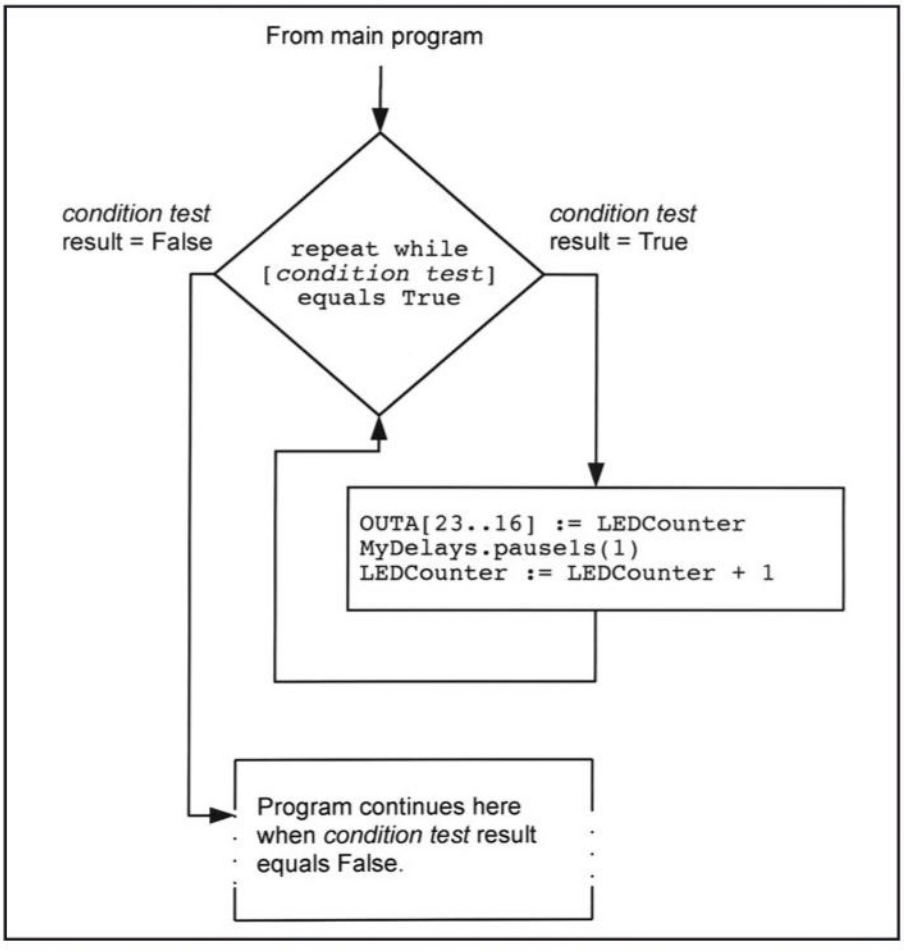


Figure 3. A repeat-while loop repeats operations while the condition test result remains True.

```
repeat until LEDCounter > 27          'Begin repeat loop
  OUTA[23..16] := LEDCounter          'Display counter value
  MyDelays.pause1s(1)                 'A 1-second delay
  LEDCounter := LEDCounter + 1        'add 1 to the count
                                      'Loop ends here
repeat                                'Do-nothing repeat loop
```

Table 2. Changes in the LEDCounter code to stop the loop from repeating once the value exceeds 27.

Loops from Start to Finish

The repeat command has another operating mode that gives program-

```
LEDCounter := 0
repeat until LEDCounter > 27          'Begin repeat loop
  OUTA[23..16] := LEDCounter          'Display counter value
  MyDelays.pause1s(1)                 'A 1-second delay
                                      'Loop ends here
repeat                                'Do-nothing repeat loop
```

Table 3. Here's what NOT to do when using a repeat until... or repeat while... command. Always update the condition-test variable within the loop or the program will never start.

```
LEDCounter := 0
repeat until LEDCounter > 27          'Begin repeat loop
  OUTA[23..16] := LEDCounter          'Display counter value
  MyDelays.pause1s(1)                 'A 1-second delay
LEDCounter := LEDCounter + 1
repeat                                'Do-nothing repeat loop
```

Table 4. This loop will also never run, because the LEDCounter := LEDCounter + 1 line was not indented and the MCU does not read it as part of the repeat sequence.

mers more flexibility when they create loops. This type of repeat statement has two general formats:

repeat *variable* **From** *start-value* **To** *finish-value*

and

repeat *variable* **From** *start-value* **To** *finish-value* **Step** *increment*

To apply either of these statements, first define a variable in the VAR section of the Propeller Tool editing window and give it a useful name. Remember to choose a variable size — byte, word, or long — based on the maximum value you expect to need. The following example uses a byte-size variable that allows as many as 256 values.

```
VAR
    byte StockNumber          'Stock numbers from 0 to 255
```

Next, decide on a start-value and a finish-value for the loop. The following example assumes the programmer intends to do something with information about products that have stock numbers 56 to 190.

```
repeat StockNumber From 56 To 190
  loop instructions here...
  and more instructions here...
```

A loop controlled by a repeat-from-to statement *automatically* increments the variable — in this case StockNumber — by 1. The loop *does not* need a statement such as:

```
StockNumber := StockNumber + 1
```

Do not add such a statement to a repeat-from-to loop and *do not* change the value of the repeat variable within the loop. Doing so could cause problems that prove difficult to troubleshoot. A programmer may use the variable to identify data, to calculate something, and so on. But these operations *must not* change the variable's value.

Suppose you want to work with information about only the even-numbered products or only every third product in the range: 56 to 190. Then a program needs a way to alter the StockNumber variable, and the Spin language includes an option so we can set the loop increment to a value other than 1. Inclusion of a step size in a repeat statement simplifies programming and automatically handles increments. The statement below illustrates how to write a repeat loop that increases the ProductNumber by 3 each time through:

```
repeat StockNumber From 56 To 190 Step 3
```

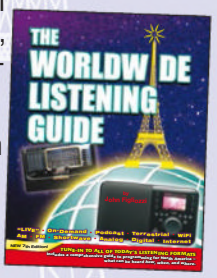

A Great Holiday Gift for the SW Listener!

NEW 7th Edition!

THE WORLDWIDE LISTENING GUIDE

by John Figliozzi

The new, expanded 7th Edition of John Figliozzi's ***Worldwide Listening Guide*** includes completely updated listings of popular radio programs that can be heard using traditional shortwave receivers, as well as today's newer listening technologies. Program listings are classified by genre and tell you the time of day and day of the week they are on-the-air, and how to find them on your shortwave receiver, WiFi radio, computer, and other listening devices.



This new edition updates and reviews all of the ways programs can be heard– "live," on-demand, WiFi, podcast, satellite, internet, digital and, of course, analog AM, FM and SW.

Spiral-bound to open in a flat, easy-to-use format. This all-new edition will be available about December 1st. Pre-order yours today!

160 pages – $24.95 + shipping

Pre-order your copy today from:

The W5YI Group
1-800-669-9594
www.w5yi.org

CQ 11-15

KJI Electronics
Customer Appreciation Days Nov. 20-22
Now an Authorized Yaesu Dealer
Full-line Dealer, Stocking:
Alinco, Icom, Kenwood
Heil, LDG, Comet, Diamond,
GAP, Palstar, SGC, ARRL, CQ,
MFJ, Ameritron, Mirage, Nifty,
Alpha/Delta ...and many more!
73s, Gene
visit www.kjielectronics.com
or the KJI Store • 973-571-1930

The NEW EZ HANG
Square Shot Kit
www.ezhang.com
Suggestions from thousands of HAM's and Cable Installers around the world, led to a complete redesign of the EZ Hang. Custom Designed for YOU, the user in mind. Now safer and easier to use, you will hit your mark every time, with less chance of misfires or hitting the yoke.

OVER 11,900 SOLD AROUND THE WORLD!
Stop by Booth 588 for the Hamvention specials!
540-286-0176
www.ezhang.com
EZ HANG
32 Princess Gillian Ct.
Fredericksburg, VA 22406

PowerPort
PowerPort-312
12 volt, 12 amp hour rechargeable battery. 3-port DC outlets,battery statics display. Padded bag with detachable accessory pockets.

831-427-8197 • KC6QLB
www.powerportstore.com

TGE
TG Electronics
N8XJK Boosters Regulators

Boost 9 Volts up to 15 Volts DC!
Boost, Filter and Regulate your DC Power!
Custom Boosters and options are available!
See our New Automatic Battery Disconnect.
Check out: www.tgelectronics.org
Call Tim @906 370-5031
Email: timig@email.com
Made in the USA

The stock number would start at 56 and then increase to 59, 62, 65, and so on up to what value? The loop would end after product number 188 because the next product number (188 + 3) would exceed the To limit of 190. Unless you specify a Step value, the Spin language assumes an automatic increment of 1.

The *repeat-from-next* statement allows for decreasing loop counts, too:

```
repeat QSO_Number From 178 To 10
```

The Spin language automatically *decreases* the *QSO_Number* variable by 1 each time through the loop until it has reached 10. This type of repeat loop will accept negative step sizes, but DO NOT use a minus sign with the step value:

```
repeat QSO_Number From 178 To 10 Step 2
```

This repeat loop will go through *QSO_Number* values from 178 to 10 this way: 178, 176, 174, and so on. Again, no minus sign for the step value. The Spin language understands how you want the loop to operate. (If you use a step value of -2, what will happen?)

A Programming Tip

Good programmers aim to eliminate hard-coded values from programs. Instead they set values as defined constants when possible. This technique simplifies documentation and debugging. Consider the following loop in this snippet of code that sends values to a display:

```
VAR
  byte :  PrintOutput

other code here...

repeat PrintOutput From 65 To 90
  Transmit PrintOutput value to a display...
  more loop commands indented here...
```

What does this loop do? We can't tell. But by defining the two constants and giving them helpful names, we get a better idea of what goes on:

```
CON
  ASCII_A = 65          'ASCII value to print letter "A"
  ASCII_Z = 90          'ASCII value to print letter "Z"

VAR
  byte :  PrintOutput

other code here...

repeat PrintOutput From ASCII_A To ASCII_Z
  Transmit PrintOutput value to a display...
  more loop commands indented here...
```

Aha. Looks like this loop sends character codes for the alphabet to some sort of display or printer. Two key points: First, nothing except a code revision can change the defined constants, and second, the use of meaningful constant names helps programmers understand what a section of code will do. Of course, comments help, too.

Coming Up in Part III …

In the next article, you will learn how the Propeller MCU gets data from external devices, and other ways an MCU makes decisions.

Notes:

1. *CQ*, October 2015.

We continue with KZ1G's tutorial on using microcontrollers for ham projects[1]. This month, we start to connect the MCU to the outside world

An Introduction to Microcontrollers, Part III

MCUs Act on Input Data

BY JONATHAN TITUS,* KZ1G

The Propeller microcontroller (MCU) has pins that can receive signals from external devices compatible with 3.3-volt logic levels. A circuit might connect an MCU with switches, a temperature sensor, a GPS receiver, and a remote-control device, for example. To use pins as inputs, a program must include a DIRA statement that identifies the pin or pins that will connect to input devices. The DIRA statement uses a logic-0 to identify such pins. For software to use pins P8 and P9 as inputs, you would write:

```
DIRA[9..8] := %00
```

After this statement sets pins 9 and 8 as inputs, an INA statement may get signal information from them. You could use an INA statement for each signal, or a single INA statement to get both signals simultaneously:

```
INA[8]   and   INA[9] , or INA[9..8]
```

What happens to the logic signals on those pins when the MCU executes an INA statement? To save this information, a program needs a variable, or variables. The snippet below defines constants for the pins used by two external input devices, assigns storage space for each input, gets the data, and saves it in those variables:

```
CON
  SensorA_Pin = 9
  GoSwitch_Pin = 8

VAR
  byte    SensorA_Data
  byte    GoSwitch_Data
```

* 14253 S. Trailview Way
Herriman, UT 84096-8029
<jontitus@comcast.net>

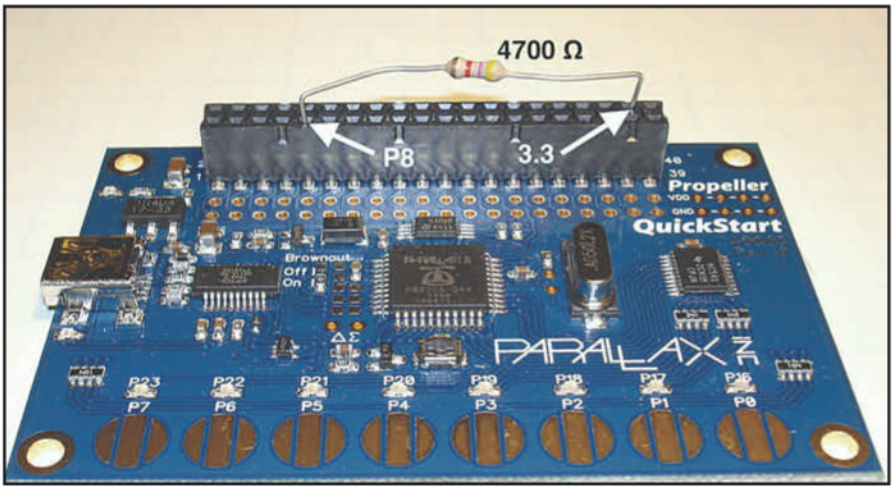


Photo A. The 4,700-ohm resistor serves to "pull up" the P8 input pin to a logic-1 level of about 3.3 volts. A circuit should not let unused input pins "float" to an unknown voltage.

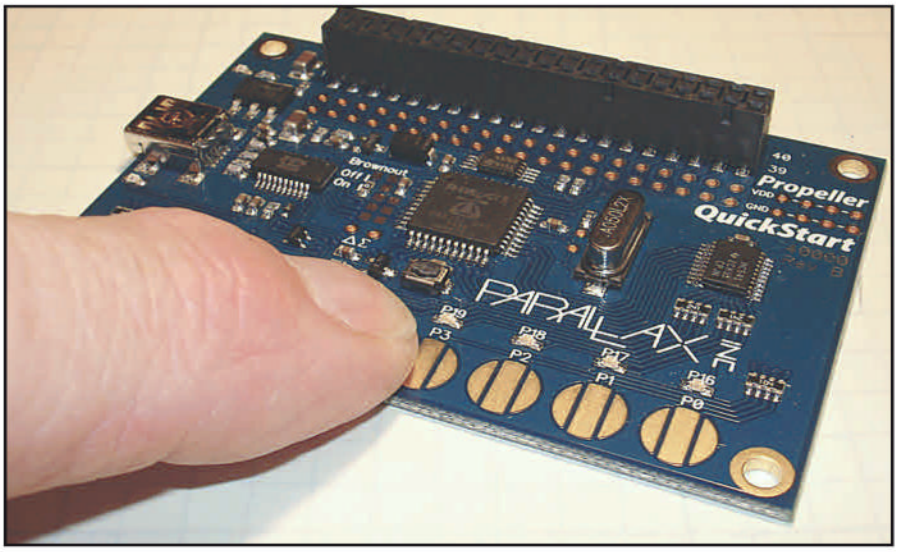


Photo B. The QuickStart board provides eight touch switches at Propeller pins P0 through P7. Software converts a signal from the touch contacts into a debounced (see text) logic-0 or -1 for your program to use.

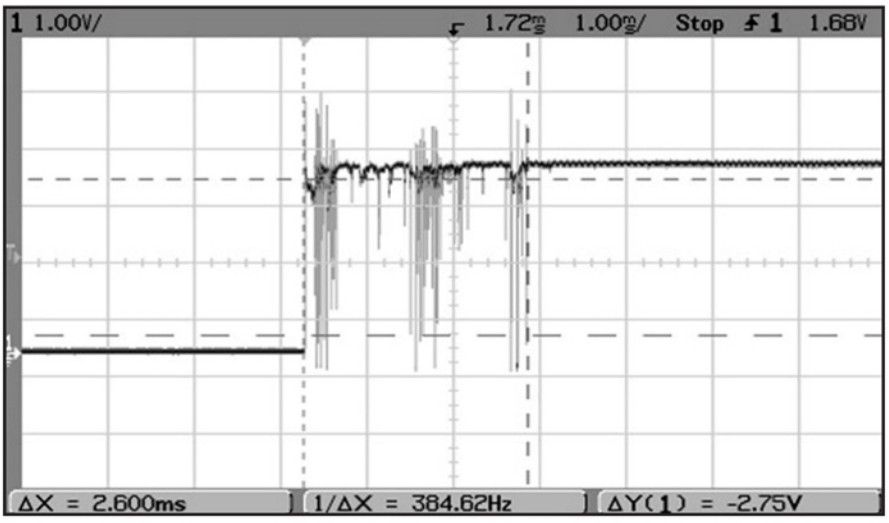


Photo C. This plot of voltage vs. time shows how a mechanical switch closure causes many bounces that result in transitions between the two logic levels for 2.6 milliseconds. MCU software might detect each bounce as a separate switch closure.

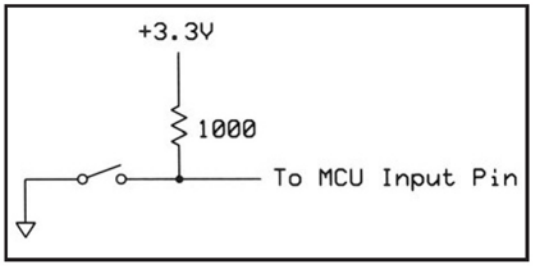


Figure 1. This switch circuit normally connects to ground. When the switch opens, a pull-up resistor provides a 3.3-V (logic-1) signal to an attached MCU. This type of mechanical switch requires "debouncing" software or hardware so the MCU detects only one logic-level transition.

```
PUB Main
  DIRA[SensorA_Pin..GoSwitch_Pin] := %00
  SensorA_Data := INA[SensorA_Pin]
  GoSwitch_Data := INA[GoSwitch_Pin]
```

Input External Information

As an alternative to saving information from an input pin, you may use it immediately. Suppose you want a logic input on pin P8 to control LED P23, and a logic input on pin P13 to control LED P21. A logic-1 input will turn on the corresponding LED, while a logic-0 turns it off. The short program below would work well:

```
CON
    _clkmode = xtal1 + pll16x     'Crystal 1 and PLL 16x
    _xinfreq = 5_000_000          'Crystal freq 5 MHz

PUB  Start                        'Program starts here
DIRA[8] := %0                     'Set P8 and P13 as inputs
DIRA[13] := %0
DIRA[23] := %1                    'Set LEDs P23 and P21 as outputs
DIRA[21] := %1

repeat                            'Unconditional repeat loop
  OUTA[23] := INA[8]              'Transfer logic signal from P8
                                  ' to LED at P23
  OUTA[21] := INA[13]             'Transfer logic signal from P13
                                  ' to LED at P21
```

Note that a DIRA[13..8] statement would set pins 13 *through* 8 as inputs, so the code above identifies them individually. Key this program into the Propeller Tool and run it on the QuickStart board. Do the LEDs turn on? Why not? The MCU needs a logic-1 signal on the P8 or P13 pin (or both) to cause the corresponding LED to turn on. You can connect a 1,000-ohm to 100k-ohm resistor between the P8 pin and the 3.3-volt receptacle (marked 3.3) on the expansion connector to create a logic-1 input (*Photo A*).

From MILLIWATTS To KILOWATTS℠
In Stock Now!
Semiconductors for Manufacturing and Servicing Communications Equipment
Visit Our Website
• RF Modules
• Semiconductors
• Transmitter Tubes
Se Habla Español • We Export
Phone: 760-744-0700
Toll-Free: 800-737-2787
(Orders only) 800-RF PARTS
Website: www.rfparts.com
Fax: 760-744-1943
888-744-1943
Email: rfp@rfparts.com
VISA MasterCard American Express Cards
RF PARTS COMPANY

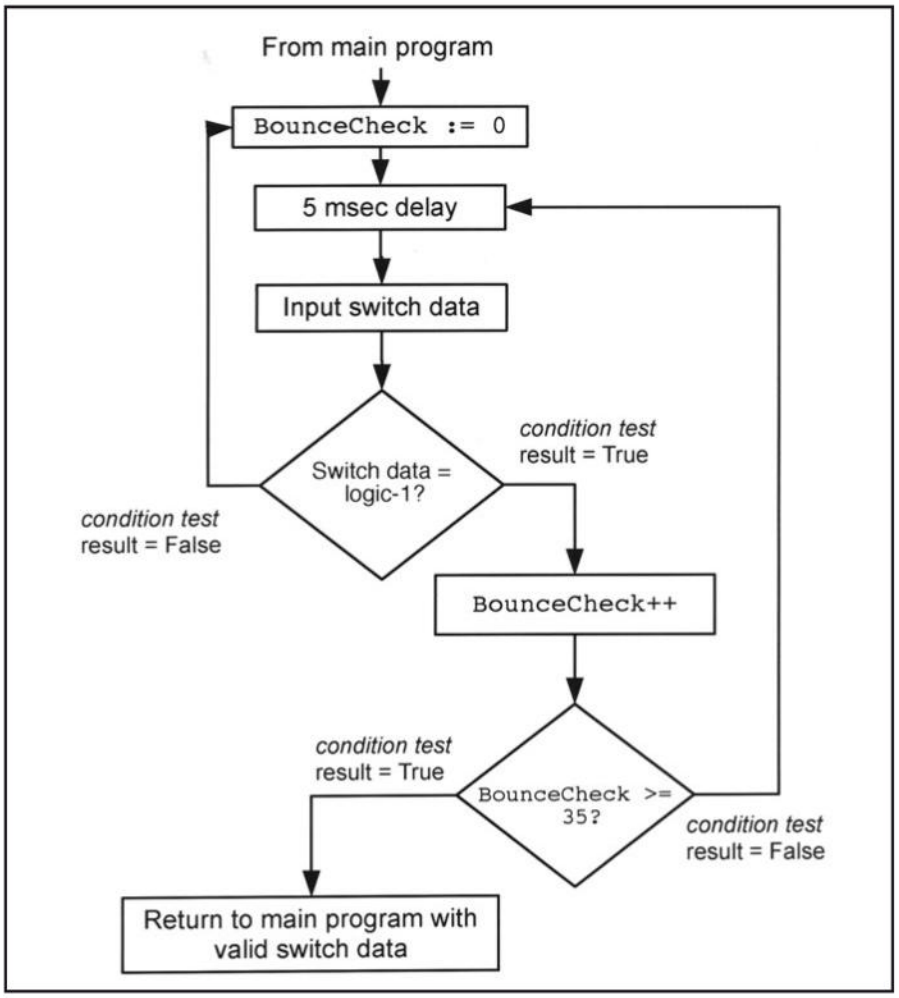

From main program
BounceCheck := 0
5 msec delay
Input switch data
Switch data = logic-1?
condition test result = True
condition test result = False
BounceCheck++
BounceCheck >= 35?
condition test result = True
condition test result = False
Return to main program with valid switch data

Figure 2. A software-debounce object includes a time delay and requires a True result for 35 consecutive logic-1 states before it reports a valid switch closure. The BounceCheck++ statement provides a shortcut for an increment-by-1 operation.

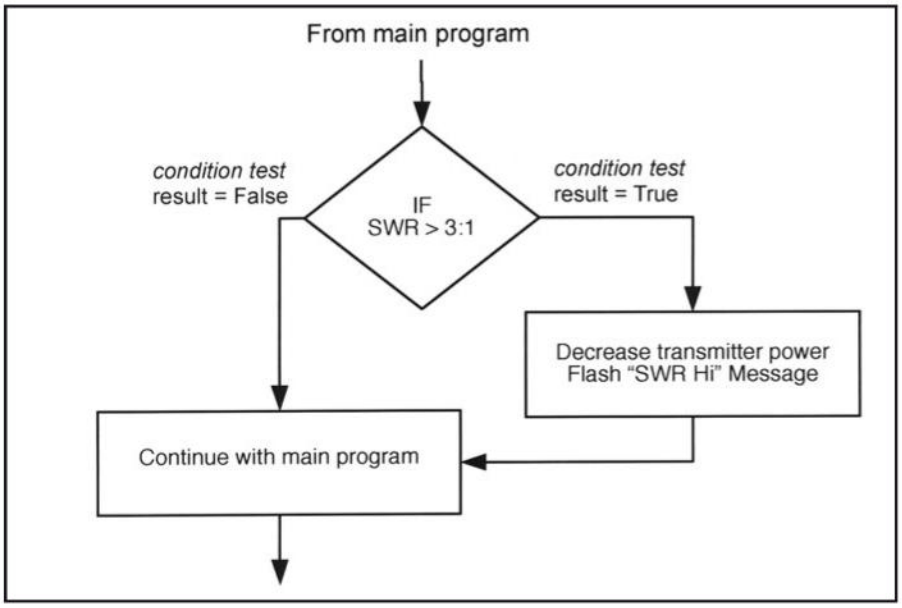

From main program
IF SWR > 3:1
condition test result = False
condition test result = True
Decrease transmitter power
Flash "SWR Hi" Message
Continue with main program

Figure 3. According to this flow chart, an MCU will decrease transmitter power and flash a message only when the measured SWR exceeds 3:1. If the SWR measures 3:1 or less, the MCU ignores the reduce-power and message operations.

Important: I highly recommend the Parallax QuickStart Pin Finder template because it simplifies MCU pin identification. Without the plastic template you must count connector pin numbers and cross reference them to the MCU I/O pin numbers. I also recommend a solderless breadboard and jumper wires for experiments with any MCU board. Many boards lack space for prototype circuits.

Get in Touch with Switches

The Propeller QuickStart board includes eight touch-switch sensors you can use immediately. These switches look like gold circles divided into three sections (*Photo B*). When a finger touches a contact, a small amount of current flows and the MCU can detect it as a closed switch. Remove the finger and the MCU detects the switch as open.

What new software do we need to use the touch switches? None. The software already exists in the Touch Buttons.spin program in the Propeller Object Exchange (OBEX). The code includes two touch-switch objects, Start and State. The first object sets a timing value for the second object, State, which reports a logic-0 or a logic-1 for each switch. Why would software for a simple switch need a timing value? It helps overcome a well-known problem called "switch bounce." When a mechanical switch completes a circuit, the switch contacts can "bounce," which causes the circuit to open and close several times. *Photo C* shows a voltage vs. time plot for a switch closure. Each time this switch closes it produces a different waveform with many transitions between logic-0 and logic-1. The number of transitions and the bounce time can vary, even for the same switch.

Equipment designers use hardware or software to overcome the problem. Of course, hardware adds cost and complexity, and something else to troubleshoot. Assume a switch normally produces a logic-0 output that changes to a logic-1 when someone actuates it (*Figure 1*). The flow chart in *Figure 2* shows how switch-debounce software would work.

The MCU sets the variable BounceCheck to 0. When software detects a logic-1 at the switch-input pin, it adds 1 to BounceCheck. The software waits a preset time and tests the pin again. (Thus the need for a timing value.) If it again detects a logic-1, the code increments the BounceCheck value. But if the software detects a logic-0, caused by contact bounce, it resets BounceCheck back to 0. Only after this routine has detected a logic-1 during 35 *con-*

secutive tests does it report a switch closure. The time between each test could range from tens of microseconds to several milliseconds. Based on tests with many of the same switch type, a programmer could specify the proper time delay.

The State object in the Touch Buttons.spin program uses a similar software-debounce technique to ensure accurate touch-switch results. The short program below illustrates how to use the Start and State objects. Whenever you touch one of the touch buttons, the corresponding LED turns on. It turns off when you remove your finger. You may run this program and test the touch switches.

```
CON
  _clkmode = XTAL1 + PLL16X          'Crystal 1 and PLL 16X
  _clkfreq = 80_000_000              'Frequency setting

OBJ

  Buttons : "Touch Buttons"          'Use "Buttons" to identify the
                                     'objects in Touch Buttons.spin
PUB Main                             'Main program starts here

  Buttons.start(_CLKFREQ / 100)      'Start the touch-button
                                     'object for sampling 100
                                     'times a second
  DIRA[23..16] := %11111111          'Set LED pins as outputs
  repeat                             'Run this loop "forever"
    OUTA[23..16] := Buttons.State    'Transfer button-state data
                                     'to the 8 LEDs
```

The program above simply transfers the button results to the LEDs, which works well as a demonstration. But to use switches effectively, you must know more about how MCUs make decisions.

"IF" Statements Alter Code Flow

You already learned how to use condition tests in loops to control their execution. The results of such tests determine when a loop would end. The Spin language includes an "IF" statement that lets programmers write decision-making code that doesn't depend on loops and counters.

An IF statement includes a test that determines whether or not the MCU executes a set of indented instructions below it. The following code snippet executes the three indented instructions, a, b, and c, only when the condition test SWR > 3 produces a True result (*Figure 3*):

```
IF SWR > 3                           'Test this condition
  a...                               'Run these 3 statements when
  b...                               'the test yields a True answer
  c...
x...
```

1. Only when the test result equals True will the MCU execute the three (or more) indented instructions that immediately follow it. The program then continues with the unindented instruction, x, below it.
2. When the test result equals False, the MCU bypasses instructions a, b, and c, and continues with instruction x.

Suppose you want to run one set of statements when the IF test yields a True result and a different set of statements when the IF test provides a False result. You add an "ELSE" statement that requires no test (*Figure 4*).

```
IF SWR > 3                           'Test this condition
  a...                               'Run these 3 statements when
  b...                               'the test yields a True answer
  c...
ELSE
  q...
  r...
x...
```

In this example a True test result causes the MCU to execute statements a through c. It skips over statements q and r indented under the ELSE statement and contin-

ROHN
Products LLC
NEVER ACCEPT SECOND BEST FOR YOUR TOWER PROJECT!
GO WITH THE ORIGINAL - DESIGNED & MANUFACTURED IN THE USA
WWW.ROHNNET.COM
DISTRIBUTORS ON THE WEB SITE
SALES@ROHNNET.COM

KJI Electronics
Now an Authorized Yaesu Dealer
Full-line Dealer, Stocking:
Icom, Kenwood, Yaesu
Heil, LDG, Comet, Diamond, GAP, Palstar, SGC, ARRL, CQ, MFJ, Ameritron, Mirage, Nifty, Alpha/Delta ...and many more!
73s, Gene
visit www.kjielectronics.com
or the KJI Store • 973-571-1930

One Separation Kit for All Popular Mobile Transceivers

Operate remotely up to 100 feet
Using SwapMyRigs Units & Cables
Radios & Tested Separation

Yaesu		Kenwood		ICOM	
FT-857	75'	TM-D710A/G	75'	ID-880H	15'†
FTM-350	75'	TM-V71	75'	IC-2720/30H	100'
FTM-100/400	90'	TS-480	100'	IC-2820H	25'
FT-7-8xxx*	100'	TM-D700	100'	ID-5100	100'
		TS-2000	100'	IC-7100	100'

* Control & APRS
† Tested with 15' VGA cable only
www.SwapMyRigs.com

ELECTRIC RADIO MAGAZINE

In circulation over 26 years, ER is a monthly publication celebrating classic equipment that was the pride of our shacks just a few years ago. Send $1 for a sample:
ER, PO Box 242
Bailey, CO 80421-0242
720-924-0171
WWW.ERMAG.COM

Post Office of the World
K3FN QSL SERVICES

Envelope Mailing Systems
International Return Postage
Custom QSL Service
Envelope Mailers:
European or Normal Size – Covers & Return
Security Tinted – No Folding of the Return
European Size Pictured Above
$5.00 per Pack plus Shipping. (25 Envelopes per Pack)
Postage:
Return Postage for Over 100 Countries – Fair Pricing
Order On-line!
K3FN Custom QSL Service:
We Will QSL for You. Choose Your Category!
Details on Our Web
K3FN@aol.com
www.airmailpostage.com

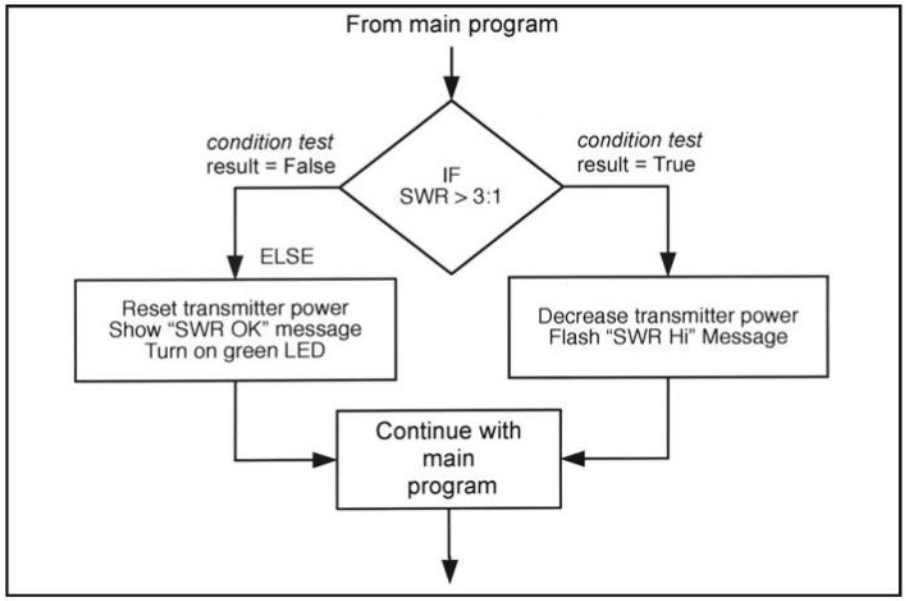


Figure 4. This flow chart shows how an IF-ELSE statements cause a program to run one set of statements for a True result and a different set of statements for a False result.

ues with statement x. A False result causes the MCU to skip over statements a, b, and c. It then execute statements q and r, and continues with statement x.[2]

I'll use three QuickStart-board touch-switch inputs and the LEDs in an example to show how an IF statement operates. Three buttons, P7, P5, and P0, will cause the LEDs to display different on-off patterns. The previous touch-switch example simply transferred touch-switch data to the LEDs. Now the software must determine the state of individual switches. How can a program do that?

Masks Isolate Bits

Programmers use a "mask" to filter, or remove, unwanted bits so code can test the state of only the bits you want. A bitwise-AND operation does the trick. *Table 1* shows the results of an AND condition with two bits, A and B. In this case, 0 represents False and 1 represents True. The only True result appears when both Input A AND Input B are True. In the other three situations, the result always comes out False. Engineers call this arrangement a *truth table*.

We may apply the same bitwise AND operation to two bytes, as seen in *Table 2*.

So we need mask bytes for switches P7, P5, and P0, and IF statements to test the bitwise-AND results. In the Spin language, a single ampersand character (&) performs the bitwise-AND operation and yields a True or a False answer:

```
CON
  _clkmode = XTAL1 + PLL16X          'Crystal 1 and PLL 16X
  _clkfreq = 80_000_000              'Frequency setting
  P7_Mask = %10000000                'Mask for switch P7
  P5_Mask = %00100000                'Mask for switch P5
  P0_Mask = %00000001                'Mask for switch P0

OBJ
  Buttons : "Touch Buttons"          'Use "Buttons" to identify the
                                     'objects in Touch Buttons.spin
PUB Main                             'Main program starts here
Buttons.start(_CLKFREQ / 100)        'Start the touch-button
                                     'object for sampling 100
                                     'times a second
```

Input A	0	0	1	1
Input B	0	1	0	1
Result	0	0	0	1

Table 1. The truth table for an AND operation shows a True output only when Input A AND Input B equal logic 1; that is, when both have a True state.

Switch Number	P7	P6	P5	P4	P3	P2	P1	P0
Switch Data	0	0	1	1	0	1	0	1
Mask Byte	0	0	1	0	0	0	0	0
Result	0	0	1	0	0	0	0	0

Table 2. A binary 1 for P5 in the Mask Byte allows only the data for switch P5 to filter through to the result. This type of mask and a bitwise AND operation lets software isolate and test a specific bit or sets of bits. Could you use this type of mask to detect a logic-0 at a specific bit? (See Note 3)

```
DIRA[23..16] := %11111111          'Set LED pins as outputs
  repeat                           'Run this loop "forever"
    Switch_Data := Buttons.State   'Save the switch information
    if Switch_Data & P7_Mask       'Test for switch P7 touched
      OUTA[23..22] := %11          'If touched, LEDs 23 and 22 on
    if Switch_Data & P5_Mask       'Test for switch P5 touched
      OUTA[17..16] := %11          'If touched, LEDs 17 and 16 on
    if Switch_Data & P0_Mask       'Test for switch P0 touched
      OUTA[20..19] := %11          'If touched, LEDs 20 and 19 on
                                   'End of loop, do it again
```

Run this program. What happens as you touch the switches? Do the LEDs turn on? Do they turn off? Groups of two LEDs turn on when you touch switches P7, P5, or P0, but they don't turn off. You could add a statement to turn the LEDs off each time through the loop:

```
repeat                             'Run this loop "forever"
  OUTA[23..16] := %00000000        'Turn all LEDs off
  Switch_Data := Buttons.State     'Save the switch information
  etc...
```

What happens then? The LEDs turn on when you touch a button and turn off when you remove a finger. But now the LEDs look dimmer. Why? The LEDs *always* get turned off by the added statement even if you still have a finger on a touch button. As a result, the LEDs always go through an on-off-on-off... sequence that makes them appear dim to our eyes.

Instead of forcing all LEDs off each time through the loop, test for the condition in which you touch NO switches and only then turn off the LEDs:

```
repeat                             'Run this loop "forever"
  Switch_Data := Buttons.State     'Save the switch information
  if Switch_Data == 0              'No switches pressed, so
    OUTA[23..16] := %00000000      'turn off all LEDs
  if Switch_Data & P7_Mask         'Test for switch P7 touched
  etc...
```

The statement:

```
if Switch_Data == 0
```

tests for the condition, "no buttons touched." (The double equal signs, ==, represent a test for equality between two values.) So if Switch_Data equals zero, the condition test yields True. The next statement turns off the LEDs, but only when you remove a finger from all touch switches. So the LEDs remain bright and turn off only when no switch gets touched. What happens if you touch all three switches, P7, P5, and P0?

Coming Up Next…

The next article will explain how an MCU communicates with external devices and how it obtains useful information from them.

Notes:

1. Parts I and II of this series appear, respectively, in the October and November 2015 issues of *CQ.*

2. For other variations of the IF-ELSE statement, refer to the online Propeller Manual, ver. 1.2, available at: <http://bit.ly/1LFJvkX>.

3. Yes. A similar AND operation with a mask of logic-1 for the bit you want to test will detect a logic-0, but first you must *invert* the states of the switch bits with the exclamation-point operator (!). This example results in a True condition for a *logic-0* from input P7: if !Switch_Data & P7_Mask. Note the exclamation point in front of the Switch_Data variable. When Alpha equals %00110110, !Alpha equals %11001001

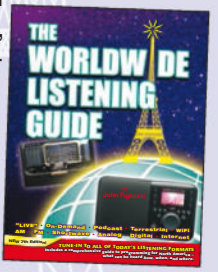
A Great Holiday Gift for the SW Listener!

NEW 7th Edition!

THE WORLDWIDE LISTENING GUIDE

by John Figliozzi

The new, expanded 7th Edition of John Figliozzi's ***Worldwide Listening Guide*** includes completely updated listings of popular radio programs that can be heard using traditional shortwave receivers, as well as today's newer listening technologies. Program listings are classified by genre and tell you the time of day and day of the week they are on-the-air, and how to find them on your shortwave receiver, WiFi radio, computer, and other listening devices.



This new edition updates and reviews all of the ways programs can be heard– "live," on-demand, WiFi, podcast, satellite, internet, digital and, of course, analog AM, FM and SW.

Spiral-bound to open in a flat, easy-to-use format. This all-new edition will be available about December 1st. Pre-order yours today!

160 pages – $24.95 + shipping

Pre-order your copy today from:

The W5YI Group
1-800-669-9594
www.w5yi.org

CQ 11-15

KZ1G continues his tutorial on how to program and use microcontrollers for various projects around the ham shack.

An Introduction to Microcontrollers Part IV

Serial Comms for Data and Devices

BY JONATHAN TITUS,* KZ1G

Circuit designers rely on bit-by-bit serial communications between microcontrollers (MCUs) and external devices because they require fewer I/O (input/output) pins than parallel data transfers. A 12-bit analog-to-digital converter would need 12 data lines and several control signals. The same converter with a serial interface might need two or three signals. Of course serial transfers take longer, but engineers often want more available I/O pins rather than faster communication rates.

Many MCUs include *universal asynchronous receiver-transmitter* (UART) hardware "blocks" that handle all control operations for bi-directional 8-bit serial communications. The word *asynchronous* refers to the lack of a clock signal between a transmitter and a receiver. MCUs also may furnish hardware blocks for other types of serial communications that require a separate clock signal.

The Propeller MCU provides no hardware dedicated to serial communications. Instead, it relies on software to gather information, properly format it, and transmit it or receive it according a given protocol. Because all 32 Propeller I/O pins operate the same way, you can assign serial I/O operations to any pins you want. You can't find that flexibility in many other MCUs.

A Parallax QuickStart board dedicates pins P31 and P30 to serial communications with a PC or other device via a USB connection. An onboard IC converts 8-bit, UART-type communications into USB signals that other devices — PCs, for example — can "understand." This IC also receives information from external devices via a USB signal and presents it to the MCU. Various programs can capture this information as one or more 8-bit "packets." *Figure 1* shows the serial format for such an 8-bit "packet." With appropriate hardware you could use Ethernet or another bus in place of USB. Serial data goes in and serial data comes out; that's what we care about[1].

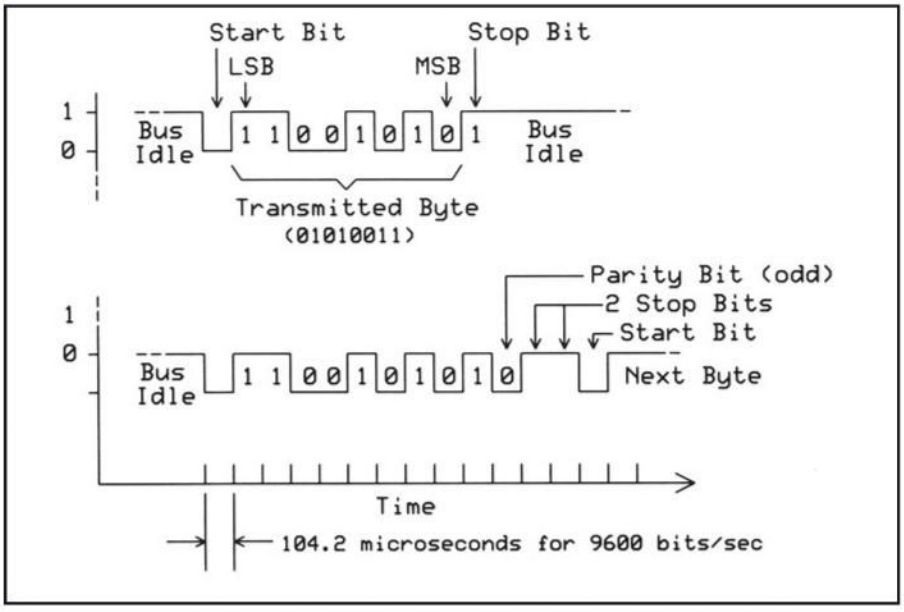


Figure 1. A standard UART-type serial communication always begins with a logic-0 start bit followed by eight data bits, a parity bit (if selected) and one or two logic-1 stop bits. The eight data bits start with the least-significant bit (LSB) and end with the most-significant bit (MSB). Newer transceivers use serial communications for rig programming and control, as well as for data transmissions via RTTY and PSK-31, for example.

The Propeller Tool software includes a terminal you can use to communicate with a QuickStart board. To begin, you must know the COM port your PC has assigned to the connected QuickStart board. Start the Propeller Tool and connect the board to your PC. Under the Run menu, click on Identify Hardware ... The PC display should indicate, "Propeller chip version 1 on COM17." Your COM number will differ from mine. Save the COM port number for your setup.

Next, go back to the Run menu, and select "Parallax Serial Terminal." A window opens and shows a white strip along the top for outgoing data and a large blue area for results from a QuickStart board. Now a Spin program may communicate with the terminal. Again, we rely on free Propeller objects to do most of the work for us.

Objects Simplify Communications

To set up 2-way communications you need programs *FullDuplexSerial.spin*

* *14253 S. Trailview Way*
Herriman, UT 84096-8029
<jontitus@comcast.net>

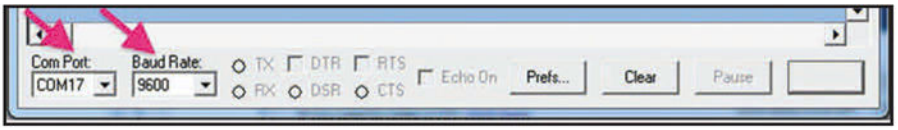


Figure 2. The terminal Com Port setting must match the port number assigned by the Propeller Tool. The Baud Rate must match the rate set in the Serial.start statement in your program.

and *Extended_FDSerial.spin* in your Propeller Workspace folder. Objects in these programs handle the details for serial communication. (I'll drop the *.spin* from now on. Unless noted otherwise, assume all program names have a *.spin* extension.)

Remember to declare use of an object this way:

```
OBJ
    Serial : "Extended_FDSerial"
```

The *Extended_FDSerial* program includes the object *start* that initializes a software-UART function and assigns a receiver and a transmitter pin:

```
Serial.start (RCVR_pin, XMTR_pin, SerialMode, Bit_Rate)
```

The *SerialMode* value configures options we don't need, so we set its value to 0. (Refer to the *Extended_FDSerial* listing for more information.) I chose *Serial* to name objects in the *Extended_FDSerial* program, but you may choose another name if you wish. This program includes another object, *str*, that will send a string of characters to the Propeller Tool terminal:

```
Serial.str(string("This is a test."))
```

Now you can write a short program to test communications with your PC. Key in the program below, or find all programs for this article at the *CQ* website <http://cq-amateur-radio.com/files/lm74_spin_progs.zip>

```
'Program Comm_Test_1.spin
CON
  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000

TransmitPin = 30          'Pin for serial output via USB to Terminal
ReceivePin  = 31          'Pin for serial input via USB from Terminal
DataRate    = 9600        'Send & receive at 9600 bits per second
SerialMode  = 0           'Normal mode, no changes

OBJ
  Serial :"Extended_FDSerial"     'Name objects for serial comms
  timer  :"Timing"                'Name objects here for a delay

PUB Start
    'Start serial port
Serial.start(ReceivePin, TransmitPin, SerialMode, DataRate)   repeat
'Repeat loop forever
   Serial.str(String("This is a test."))          'Transmit this string
   timer.pause1ms(200)    'Delay 0.2 seconds
```

Save this program in your Propeller Workspace folder as *Comm_Test_1*. Start the program (Run→ Compile Current→Load RAM) and switch to the terminal. Then go to the small Com Port window at the bottom-left corner of the terminal window and select the port assigned to your QuickStart board (*Figure 2*). Next, select 9600 in the Baud Rate window.

In the blue terminal window you should see the text, "This is a Test." displayed again and again without the quote. You probably want a new line for each sentence. The *Extended_FDSerial* program includes a way to send a "carriage-return" command (ASCII 13_{10}) to the terminal. You won't see "13" in the text window, though. The terminal interprets the value as a command in the following use of the Tx object.

From MILLIWATTS To KILOWATTS℠
In Stock Now!
Semiconductors for Manufacturing and Servicing Communications Equipment
Visit Our Website
• RF Modules
• Semiconductors
• Transmitter Tubes
Se Habla Español • We Export
Phone: 760-744-0700
Toll-Free: 800-737-2787
(Orders only) 800-RF PARTS
Website: www.rfparts.com
Fax: 760-744-1943
888-744-1943
Email: rfp@rfparts.com
RF PARTS COMPANY

what's new

Anderson Power Products Introduces the SBS Mini, DC Power Connector

Anderson Power Products has introduced the SBS Mini, the company's smallest DC power connector in its SB product group. Since 1953, the SB line has used the flat wiping contact technology, which was carried over to the popular Powerpole connector with single position stackable housings. SBS Mini simplifies the assembly process by offering a one-piece housing that holds two Powerpole 15/45 contacts.

Sacrificial contact tips designed into all SB and Powerpole connectors enable hot swap capabilities on DC circuits. The low resistance Powerpole contacts accept 20 to 10 AWG (0.75- to 4-mm) wires allowing up to 52 amps of UL rated performance per position.

SBS Mini adds additional insulation to the mating interface to protect against accidental exposure to live circuits. Easy-to-grip housings inherit color-coded mechanical keys from the SB connector. Each key is assigned a unique color for quick circuit identification. This is of particular importance when distinguishing series from parallel connections, or circuits operating at incompatible voltages.

The SBS Mini has a suggested retail price of $0.98 and is available right now. For more information regarding SBS Mini, contact: Anderson Power Products 13 Pratts Junction Road, Sterling, MA 01564-2305. Phone: (978) 422-3600. Fax: (978) 422-3700. Website: <www.andersonpower.com>.

Note: *"What's New" is not a product review and does not constitute a product endorsement by* CQ. *Information is primarily provided by manufacturers/vendors and has not necessarily been independently verified.*

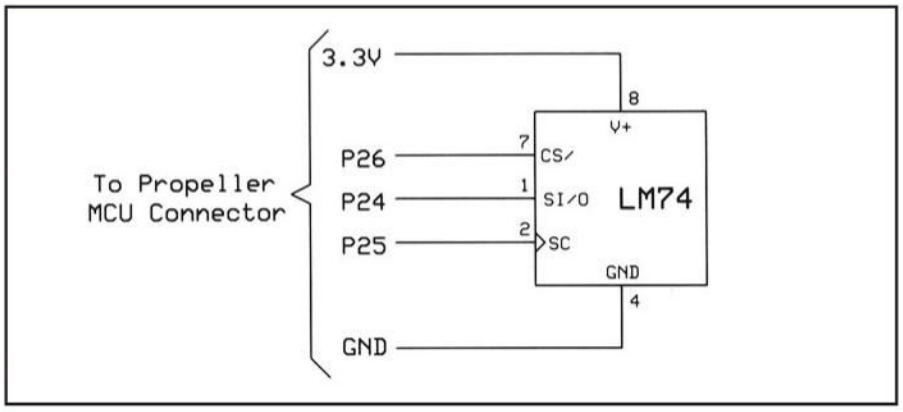


Figure 3. An LM74 temperature sensor requires three control signals from an MCU in addition to power and ground. The SC input accepts a clock from an MCU and the SI/O connection can transmit or receive sensor information. The Chip-select signal (CS/) controls the readout of temperature values.

```
Serial.Tx(13)
```

Other values print alphanumeric characters or symbols. The American Standard Code for Information Interchange (ASCII) also includes 32 other control codes[2]. You transmit a decimal value for the terminal to display this way:

```
Serial.dec(12395)
```

In this case, the MCU sends the ASCII data for a numeral 1 followed by the ASCII data for numeral 2, and so on. Save the *Comm_Test_1* program in your Propeller Workspace. Then add the lines shown in bold in the *partial* listing that follows for the *Comm_Test_1* program. *This listing is not complete*. Save this program in your workspace folder with the name *Comm_Test_2*. Now run it to confirm the changes work. Each "This is a test." should appear on a new line with an increasing count.

```
VAR
  word counter     'Set up a word-size variable

PUB Start
    'Start serial port
Serial.start(ReceivePin, TransmitPin, SerialMode, DataRate)   counter := 0
'Clear counter to zero
repeat    'Repeat loop forever
  Serial.str(String("This is a test."))
  Serial.dec(counter)        'Print the decimal count value
  Serial.Tx(13)              'Cause a carriage return
  counter++                  'Increase counter value by 1
  timer.pause1ms(200)        'Delay 0.2 seconds
```

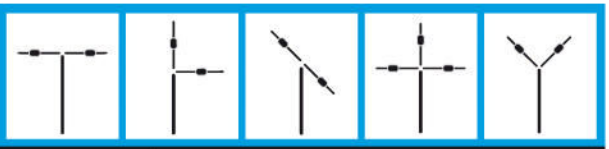
Earlier I noted you need the program *FullDuplexSerial* in your Propeller Workspace, but it doesn't appear in the OBJ section of either program. Our code does not directly use any objects in the *FullDuplexSerial* program, but the *Extended_FDSerial* program does. If you fail to include a needed program in the Propeller Workspace folder, the Propeller Tool will notify you.

You also can use objects in *Extended_FDSerial* to receive information from the PC keyboard via the terminal software. The statement:

```
VAR
    word  EnteredValue
...
EnteredValue := Serial.RxDec
```

accepts a decimal value from the terminal keyboard and assigns it to variable *EnteredValue*. You also can key in values and strings of characters. To see how

BUDDIPOLE



NEW RIGID SERIES ADJUSTABLE WHIP ANTENNAS

These combine the ruggedness of our military-grade shockcord elements with the adjustability of a telescopic antenna to provide a truly no-compromise field antenna. Available in multiple configurations.

BUDDIPOLE FEATURES

- Multi-band design works 9 bands (40 meters thru 2 meters) with one set of adjustable coils!
- Modular Design – create dozens of different antennas with interchangeable parts
- Rotatable/Directional
- Lightweight, rugged components
- Rotating Arm Kit allows users to instantly change antenna configurations
- Used by Emergency Services Groups throughout the world
- Rated from QRP to 250 watts PEP

WHAT IS THE BUDDIPOLE?

The Buddipole™ Portable Dipole fits in your travel bag and assembles in minutes. The Buddipole is more than an antenna, it's a versatile system for launching your signal. Optimized for transmit power and proven for DX work, the Buddipole is the secret weapon used by HF portable operators all over the world.

Please call or write for free brochure!

3028 SE 59th Ct. #600
Hillsboro, OR 97123

tel: (503) 591 8001
fax: (503) 214 6802

info@buddipole.com
www.buddipole.com

See our videos
www.youtube.com/buddipole

a keyboard input works, run program *Comm_Test_3*, provided in the downloaded file.

Talk with a Sensor

Bit-by-bit transfers let MCUs communicate with components that use an Inter-Integrated-Circuit (I2C) bus, a Serial Peripheral Interface (SPI) bus, a 1-Wire bus, and others. The LM74 temperature sensor, for example, uses a 2-wire SPI interface but it needs an additional wire for a chip-select signal, as seen in *Figure 3*[3]. Of course, the sensor also needs power and ground connections.

The LM74 comes in two versions: The LM74CIM-3 that operates with a 3- to 3.6-volt supply, and the LM74CIM-5 that requires from 4.5 to 5.5 volts. I use the 3-volt version because the Propeller MCU uses the same voltage[4]. Unfortunately for experimenters, the LM74 IC comes in a small-outline IC (SOIC) package that doesn't plug into a solderless breadboard. I use Schmartboard SOIC-to-DIP adapters (part no. 204-0004-01) that simplify soldering and come with pins that plug into breadboards[5]. *Photo A* shows an LM74 IC connected to a Propeller MCU.

The *LM74_Demo* program that follows displays *hexadecimal*, or base-16, temperatures in the terminal window so you can test the sensor circuit. Before you test the program, ensure you have the following programs in your workspace folder. You may have other programs in the folder, too:

- LM74_Demo
- LM74_Objects
- shiftio
- timing
- FullDuplexSerial

Run the program on your QuickStart board and switch to the terminal window. Click on the Enable control (lower-right corner) and you should see 4-digit hex values appear one after another at 3-second intervals. Heat the sensor slightly with a finger or a hot-air hair dryer (low setting). Cool the sensor with a double-bagged ice cube. *Table 1* lists Celsius temperatures and the sensor's raw binary and hexadecimal values.

The final version of a Spin program often has a hierarchy with several "levels" of objects, as shown in *Figure 4* for the next program, *LM74_Demo_2*. The diagram illustrates the beauty of Spin-language objects; they're easy to use and many portions of a program can reuse them regardless of use elsewhere in a program.

The *LM74_Demo_2* program uses two objects, *start* and *gettempc* in the *LM74_Objects* program. The *start* object initializes the LM74, puts it in a continuous-measurement mode, and sets the variable *started* to True. True is a predefined constant, as is False. The *gettempc* object uses a *shiftin* object to get 16 bits of sensor information. The *gettempc* object also tests the *started* flag to ensure software initialized the sensor. Programmers use flags to indicate a True or False condition to other parts of a program.

If you try to use the *gettempc* object with the *started* flag set in the False state, this object will not get data from the sensor. Instead, it returns a "temperature" of hex FF81, or -127°C. The sensor should not experience this temperature because it has a lower operating limit of -55°C. The FF81 (-127°C) simply serves as another type of flag that indicates the object *gettempc* tried to access the sensor but the sensor was never initialized.

Within the *LM74_Demo_2* program, the code compares the reported temperature, *TempC_Data*, with FF81. If equal, the

HamRadio Anthologies

BLOW OUT SALE!

Enjoy collections of the best material published in HamRadio magazine, conveniently arranged by subject and original publication date. Choose your interest, your time period, your anthology!

Homebrewing Techniques

This anthology brings together the most useful and practical advice and techniques for the person who wants to build anything from small solid state projects to beam antennas.

Order AHOME ~~$16.00~~ $8.00

Test Equipment & Repair Techniques

From building test gear to trouble shooting the rig, this anthology of the best articles on the subject has been carefully selected to meet today's needs. Techniques and devices that work and are easily duplicated, and gives today's Hams a much-needed helping hand at solving equipment problems.

Order ATEST ~~$16.00~~ $8.00

Ham Radio Anthology: Antennas

These two antenna anthologies cover all types of antenna designs and theory from 160 meters through microwaves. Articles have been selected to be as timely and valuable to today's Ham.

Antennas - 1968-1972

Order No. ANT1 ~~$16.00~~ $8.00

Antennas - 1973-1975

Order No. ANT2 ~~$16.00~~ $8.00

Buy all 4 for only ~~$60~~ $30 Order 4ANTS

S & H: US - add $7 for first book, $3.50 for second and $2 for each additional. CN/Foreign-$25 for first iteml, $10 for second item, $5 for each additional item.

CQ The Radio Amateur's Journal

17 West John Street
Hicksville, New York 11801
Phone 516-681-2922 • FAX 516-681-2926
http://store.cq-amateur-radio.com

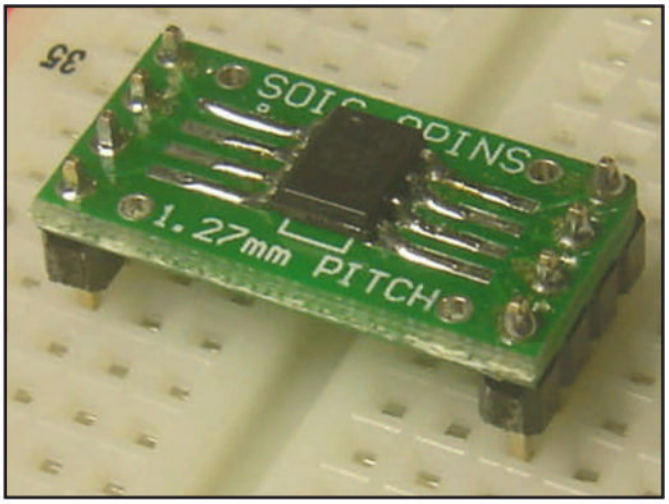


Photo A. An 8-pin Schmartboard SOIC adapter lets designers and experimenters plug an LM74 temperature sensor into a solderless breadboard with receptacles on 0.1-inch (2.54-mm) centers. Schmartboard created a technique that simplifies soldering SOIC packages to a circuit board.

Temperature (°C)	Digital Output Binary	Hex
+150	0100 1011 0000 0111	4B07
+125	0011 1110 1000 0111	3E87
+25	0000 1100 1000 0111	0C87
+0.0625	0000 0000 0000 1111	000F
0	0000 0000 0000 0111	0007
-0.0625	1111 1111 1111 1111	FFFF
-25	1111 0011 1000 0111	F387
-55	1110 0100 1000 0111	E487

Table 1. Representative temperatures and equivalent LM74 output data in binary and hexadecimal formats.

MCU displays "LM74 Not Enabled." Then the code keeps the MCU in an endless *repeat* loop until you turn it off or reprogram it. This type of error test can prevent errors under unexpected conditions, although a real-world program would not keep an MCU in an endless do-nothing loop!

Format Temperature Data

Instead of hexadecimal values, we want Celsius temperatures, so we must understand how to interpret "raw" LM74 sensor data. *Table 2* shows the arrangement of bits received from an LM74. The most-significant bit (MSB) at D15 represents the sign of the temperature. A logic-0 indicates a positive temperature and logic-1 indicates a negative temperature. Bits D14 through D3 give us a 12-bit temperature value. Bits D2 through D0 have no meaning, so we ignore them.

To create a negative binary value you **cannot** simply set bit D15 to a 1. The sensor data arrives in **2's-complement** format. To simplify a discussion of 2's-complement data, I'll use 8-bit values. The MSB (D7) will represent the sign, and bits D6-D0 represent a value. Eight-bit, 2's-complement values range from -128 to +127, or 10000000_2 to 01111111_2.

Say we have the binary value 12_{10}, (00001100) and we need a -12_{10} for a calculation. *Table 3* shows the four steps to yield a 2's-complement -12 in binary format.

The first step starts with 12_{10} as a binary number: 00001100_2. In Step 2, we complement the value bit by bit. An original 1 becomes a 0, and a 0 becomes a 1. Step 3 has

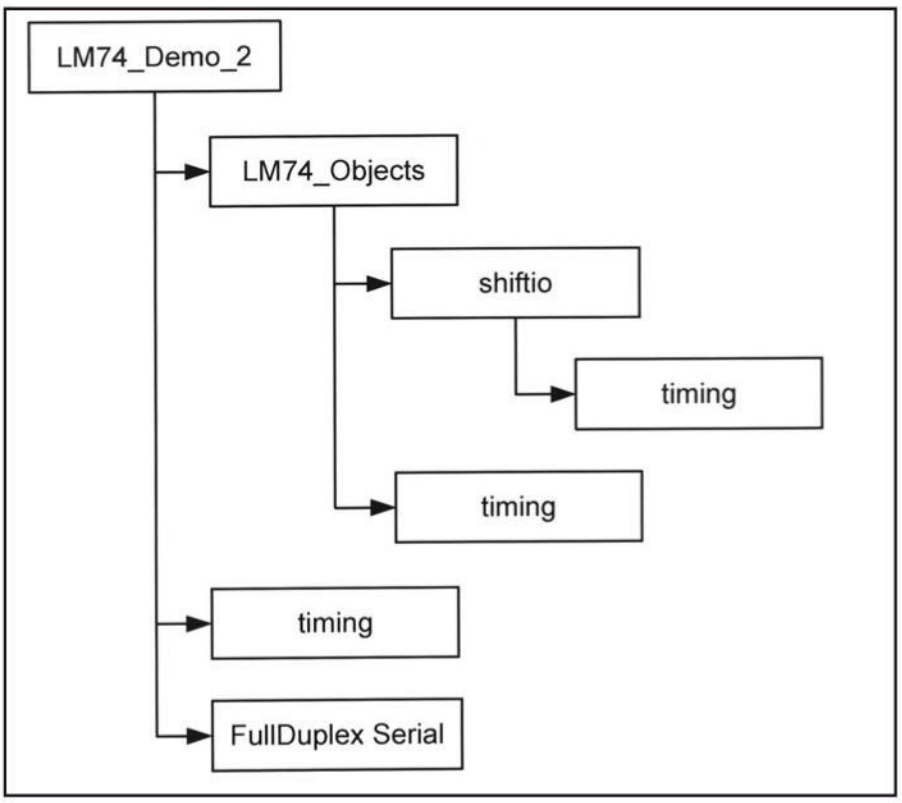


Figure 4. This hierarchy tree shows the use of Spin objects by the main program LM74_Demo_2. The timing object gets used three times in different parts of the software.

us add a 1 to the complement value to yield 11110100_2. The sum represents -12 in Step 4. In the Spin language, the exclamation-point operator "!" performs a bit-wise complement of a value.

To prove 11110100_2 equals -12_{10}, add the original +12 value (00001100_2) to the 2's complement value:

```
  00001100  =  +12
+11110100  =  -12
 100000000  =    0
```

We discard the leading 1 in the answer, an overflow from the addition, because we can't save a 9-bit value in an 8-bit byte. The result equals 0. For a binary-addition tutorial, visit: <http://academic.evergreen.edu/projects/biophysics/technotes/misc/bin_math.htm>.

Suppose we have a value such as 11001010_2 from a sensor that produces 2's-complement data. How do we convert it to a useful value we understand? First, the MSB indicates a negative value, so write down a minus sign. Second, use the steps in *Table 3* to get the 2's complement of the **entire** value, including the sign bit:

11001010_2 then becomes 00110110_2 = 54

Remember to include the minus sign, so you have -54 as the final answer. Try the 2's-complement conversion for the -25°C binary value from an LM74, but without bits D2-D0, which aren't part of the 12-bit temperature value. From *Table 1*, -25°C equals (spaces added for clarity):

1111 0011 1000 $0XXX_2$ or 1 1110 0111 0000_2 without bits D2-D0.

Take the complement of the original value — again, including the sign bit — to get: 0 0001 1000 1111_2 and then add 1.The result comes to: 0 0001 1001 0000_2. In decimal, that equals 410! What happened? Remember, the LM74 reports temperatures in units of *1/16th of a degree*, not in whole degrees (see *Table 2*). So divide 410 by 16 and you get the integer value 25. Remember the minus sign for

YAESU
FT-991

The **Yaesu FT-991** does it all. HF to 70cm with 100W on HF/6m. Multimode including C4FM Digital, HF/6m tuner, stunning 3.5" color TFT touch panel with spectrum display.
FTDX1200

The **FTDX1200** provides sophisticated 160 to 6 meter operation with the latest 32 Bit high speed DSP technology. With three roofing filters, TFT color display, waterfall scope, built-in antenna tuner, IF shift/notch & more! Visit **www.universal-radio.com** for details.
universal radio inc.
Universal Radio
6830 Americana Pkwy.
Reynoldsburg, OH 43068
◆Orders: 800 431-3939
◆Info: 614 866-4267
www.universal-radio.com

The NEW EZ HANG
Square Shot Kit
www.ezhang.com

Suggestion from thousands of HAM's and Cable Installers around the world, led to a complete redesign of the **EZ Hang**. Custom Designed for YOU, the user in mind. Now safer and easier to use, you will hit your mark every time, with less chance of misfires or hitting the yoke.
OVER 12,400 SOLD AROUND THE WORLD!
See us at Orlando Hamfest.
540-286-0176
www.ezhang.com
EZ HANG
32 Princess Gillian Ct.
Fredericksburg, VA 22406

PowerPort
WORLD PACK II

Put your multi band radio on your back with lithium power.
Heavy duty, padded, water resistant construction. Contoured shoulder straps, multiple attachment points, separate battery compartment.
831-427-8197 • KC6QLB
www.powerportstore.com

Bit Position	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
Bit Weight (°C)	Sign	128	64	32	16	8	4	2	1	1/2	1/4	1/8	1/16	X	X	X
Binary Value	32768	16384	8128	4096	2048	1025	512	256	128	64	32	16	8	4	2	1

Table 2. Format for data from an LM74 sensor. Note the "weight" of each bit in a temperature reading and the different binary weight in a 16-bit value. The LM74 measures temperature to a 1/16th of a degree Celsius.

MSB						LSB		
Step Number	D7	D6	D5	D4	D3	D2	D1	D0	Data Actions
1	0	0	0	0	1	1	0	0	Binary value for decimal 12.
2	1	1	1	1	0	0	1	1	Invert each bit to create a 1's complement binary value.
3	0	0	0	0	0	0	0	1	Add 1 to one's complement value of 12 in the row above.
4	1	1	1	1	0	1	0	0	Sum of 1 plus 1's complement of binary 12, which equals -12.

Table 3. Steps necessary to create a negative 2's-complement value from a positive value.

a final -25°C temperature. Now let's see how the Propeller MCU handles these operations in software. The flow chart in *Figure 5* shows the operations, as explained below:

1. The D2-D0 bits in the LM74 information have no use, so the program uses a bitwise-AND operation to force all three bits to logic-0. Always good to put unused bits in known state.

2. Check for a negative temperature. The *binary weight* for bit D15 is 2^{15}, or 32768. Thus if D15 = 1, the binary value for the entire 16-bit word *must* equal or exceed 32768. An `if` statement with a comparison test handles this task. For now, look at the temperature data as a 16-bit value and don't think of bits having temperature the "weights."

a. If we detect a negative temperature value, display a minus sign and get the 2's complement of the value.

b. If we don't detect a negative value, display a plus sign. We do not need the 2's complement of a positive temperature.

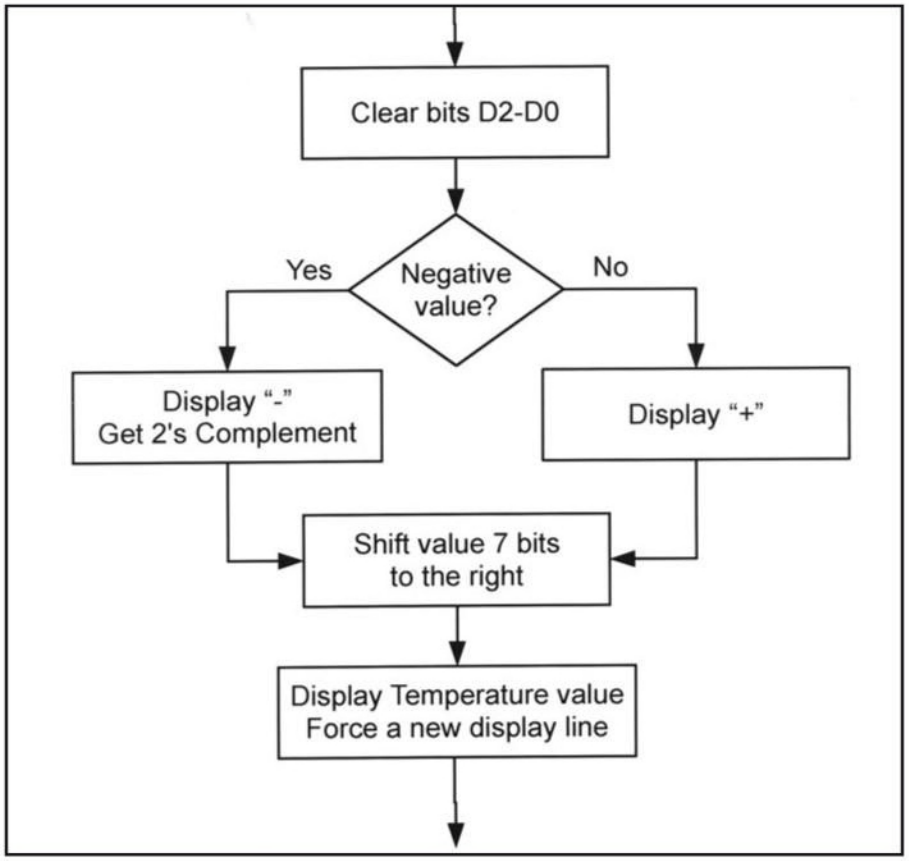


Figure 5. A flowchart for the operations that convert LM74 temperature data into temperatures humans will understand.

3. A bitwise shift-right operation removes the D2-D0 bits (three shifts right) and the additional four shifts right perform a binary divide-by-16 operation. A bitwise shift operation does not save any "shifted-out" bits. They no longer exist.

4. Display the temperature value and move the terminal cursor to a new line.

Program *LM74_Demo_2* provides code that performs these steps. Even if you won't use it, I encourage you to look at it so you see the statements that correspond to the steps above.

Food for Thought

The LM74 temperature data has a maximum accuracy of about ±1°C between -10°C and +65°C. Could you devise a way to use the bit that represents 1/2°C (D6 in the received 16-bit data) to round the temperature to the nearest degree? How would you do it?

Notes:

1. You will see UART serial-communication specifications noted as "8-N-1" or "8-E-2," for example. The first group indicates eight data bits, no parity, one stop bit. The second group indicates eight data bits, even parity, and two stop bits. Most equipment uses an 8-N-1 setup. Data rates range from 110 bits/second (bits/sec, or bps) to 115,200 bits/sec, although some devices offer higher rates. For more details, visit: <https://learn.sparkfun.com/tutorials/serial-communication>.

2. <https://en.wikipedia.org/wiki/ASCII>

3. For information about the SPI bus, visit: <https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus>

4. Download an LM74 data sheet at: <http://www.ti.com/lit/ds/symlink/lm74.pdf>

5. For Schmartboard information, visit: <http://schmartboard.com/>

KZ1G continues his ongoing series on the basics of using microcontrollers in your ham projects.

An Introduction to Microcontrollers, Part 5

Run Programs in Eight MCUs

BY JONATHAN A. TITUS,* KZ1G

Experimenters and engineers often think, "I wish I had a second microcontroller for this project." Take heart, one Propeller chip provides *eight* MCUs (*Photo A*), However, our examples have used only one. Let's suppose a circuit must perform three actions at different times: Action A every 1.2 seconds, Action B every 0.482 seconds, and Action C every 3.37 seconds. You might need three independent timers; one for each action. Even when an MCU includes many timers, writing code to handle them can present a challenge.

When you use a Propeller chip, you could assign three MCUs to handle the three tasks. We call these stand-alone MCUs "cogs," and so far, our example programs have used only one cog, Cog 0. Objects we copied from the OBEX might have used more cogs, though. We just don't realize it unless we carefully examine their code.

The eight cogs rely on a "Hub" you can think of as an 8-position rotary switch that cycles through each cog in a round-robin fashion. According to the Propeller manual, "The Hub, and the bus it controls, runs at half the System Clock rate. This means that the Hub gives a cog access to *mutually exclusive resources* once every 16 system-clock cycles." Those resources comprise Main memory and several special registers. For an 80-MHz system-clock frequency set up by the statements below, each cog can access the Main memory every 200 nanoseconds (nsec):

```
CON
  _clkmode = xtal1 + pll16
  _xinfreq = 5_000_000
```

Figure 1 shows the two types of memory in a Propeller chip. The Main memory comprises a section of read-write memory (RAM) and one of read-only memory (ROM). All cogs may access locations in Main memory as 8-bit *bytes*, 16-bit *words*, or 32-bit *longs*[1].

The Hub selects cogs sequentially from Cog-0, Cog-1, ...Cog-7, Cog-0, Cog-1 …, and so on. That means only one cog may access Main memory at a time, which avoids a conflict when, Cog-1 and Cog-5 both need to access the same memory location or variable. If a cog doesn't need to access memory it can continue to run its code without waiting for the Hub to select it.

Each cog has its own registers, 496 longs for general-purpose use and 16 longs for special purposes. No other cog may access these 512 registers. And a cog may access its registers only as long values. No byte or word accesses are allowed.

Run a Multi-Cog Program

The example program, **LED_MultiFlash**, lets you observe different flash rates and durations of three LEDs at P16, P18, and P20 on a Propeller QuickStart board (P8X32A). Download this program and others at: <http://bit/ly/

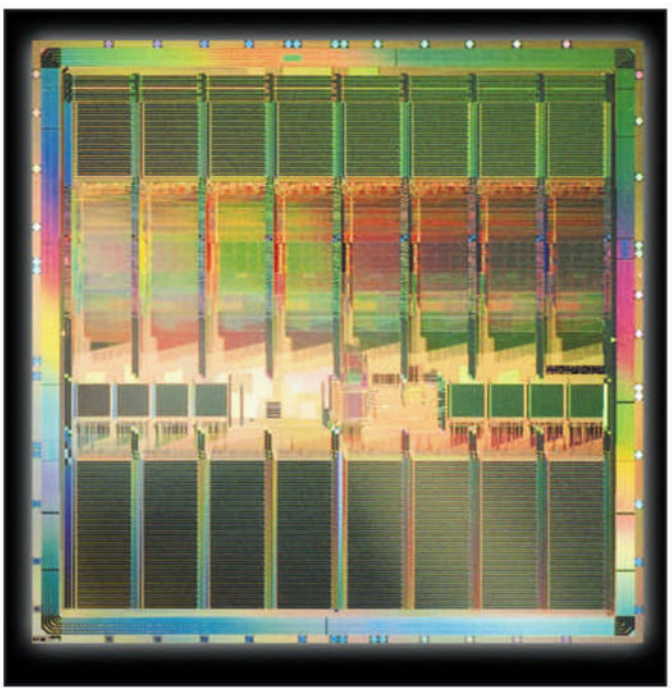


Photo A. This photomicrograph of a Propeller chip shows eight cogs and registers across the top and eight blocks of RAM along the bottom. The eight small blocks in the center hold the ROM used for a character set, math values, and a bootloader. Other areas include buses and logic circuits. (Courtesy of Parallax, Inc.)

** 14253 S. Trailview Way*
Herriman, UT 84096-8029
email: jontitus@comcast.net

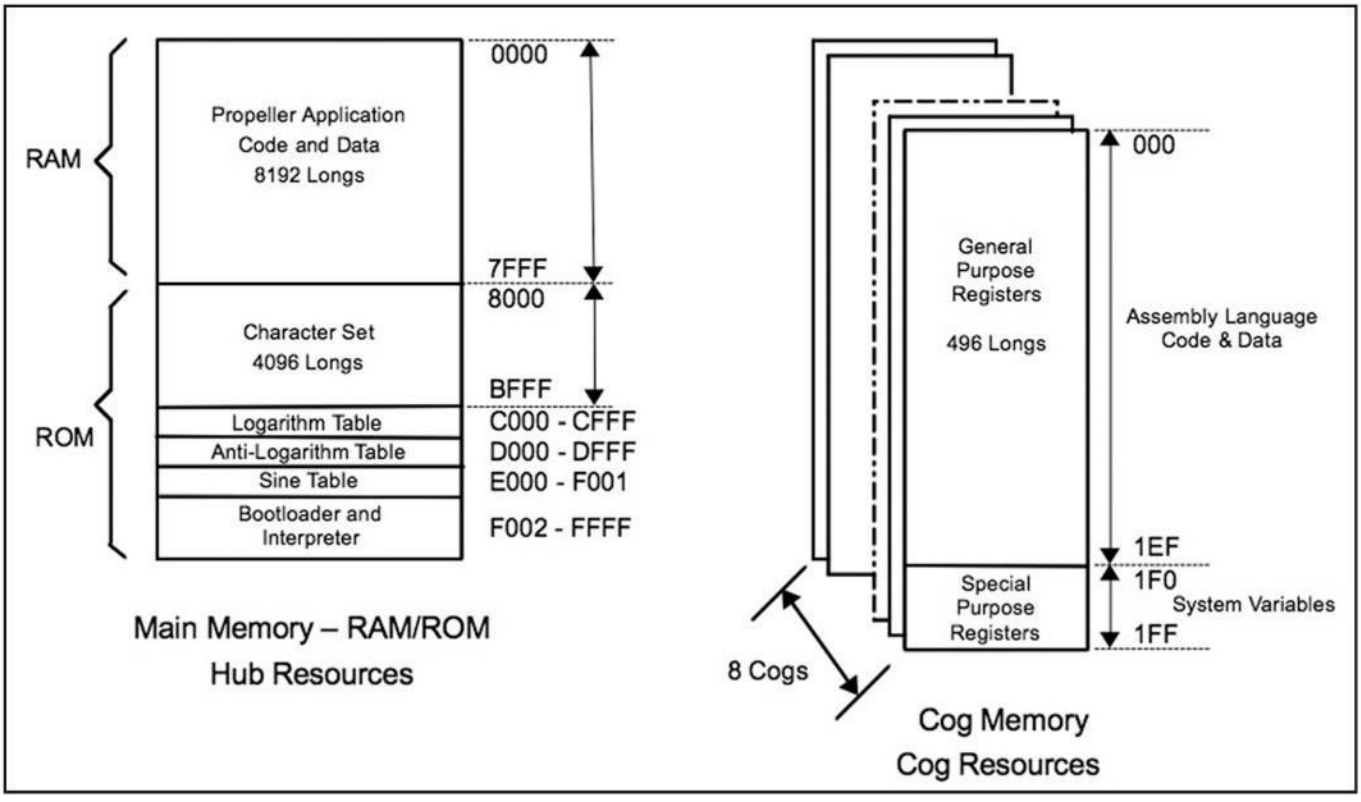


Figure 1: The memory "map" for a Propeller IC includes Main memory; RAM, and ROM (left) contained in the Propeller Hub. Each cog has a set of registers (right) for its exclusive use.

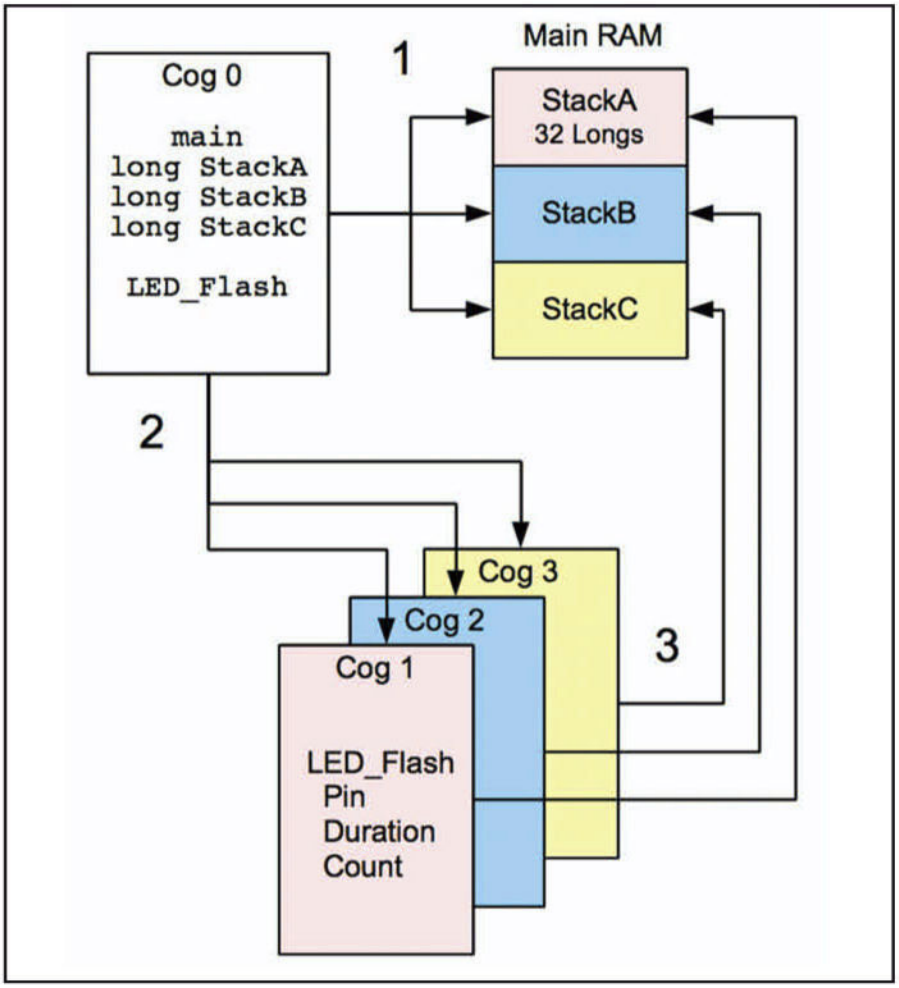


Figure 2: This diagram illustrates three operations in the LED_MultiFlash program that uses three cogs. At point 1, the program defines three arrays in RAM, one for each cog. At point 2, the main program loads the LED_Flash code into three separate cogs. This process identifies the stack location (3) created for each cog and it "sends" three values, Pin, Duration, and Count to each cog.

1m10dlG>. Run the program and watch the LEDs. Can you see the different flash rates? The software used three cogs, one per LED.

How does the program set up three cogs and run software in them? *Figure 2* shows what happens when the main program starts. To begin, each cog needs some space in the Hub's Main-memory RAM to temporarily store information. We call this area a *stack* and create it as an array of longs with 32 longs assigned to each cog. You can define a larger or smaller stack depending on the space a program needs for temporary storage. I will delve a bit more about stack size later. An array structure lets a cog easily refer to locations with an index number such as StackA[0] or StackA[22].

The LED_MultiFlash program listing (*Figure 3*) provides more details. The

HamRadio Anthologies

Enjoy collections of the best material published in HamRadio magazine, conveniently arranged by subject and original publication date. Choose your interest, your time period, your anthology!

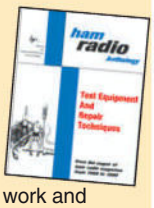
Homebrewing Techniques

This anthology brings together the most useful and practical advice and techniques for the person who wants to build anything from small solid state projects to beam antennas.

Order AHOME ~~$16.00~~
NOW JUST $8.00

Test Equipment & Repair Techniques

From building test gear to trouble shooting the rig, this anthology of the best articles on the subject has been carefully selected to meet today's needs. Techniques and devices that work and are easily duplicated, and gives today's Hams a much-needed helping hand at solving equipment problems.



Order ATEST ~~$16.00~~
NOW JUST $8.00

Ham Radio Anthology: Antennas

These two antenna anthologies cover all types of antenna designs and theory from 160 meters through microwaves. Articles have been selected to be as timely and valuable to today's Ham.

Antennas - 1968-1972
Order No. ANT1 ~~$16.00~~
NOW JUST $8.00

Antennas - 1973-1975
Order No. ANT2 ~~$16.00~~
NOW JUST $8.00

Buy all 4 ~~$80~~
NOW JUST $30 Order 4ANTS

S & H: US - add $7 for first book, $3.50 for second and $2 for each additional. CN/Foreign-$25 for first iteml, $10 for second item, $5 for each additional item.

CQ The Radio Amateur's Journal
17 West John St. • Hicksville, NY 11801
Phone 516-681-2922
FAX 516-681-2926
http://store.cq-amateur-radio.com

software in *Figure 3* first creates three memory arrays, StackA, StackB, and StackC, and each array can store 32 longs. The Main part of the program uses a cognew statement to start a new cog and load it with the LED_Flash public object located below the Main section of the program. The loading process transfers three parameters — *Pin, Duration, and Count* — to the cog along with a stack *address* (@Stack...) This address lets the cog identify its storage area in memory. The cog starts to run almost immediately. The Main section of code continues to perform the same actions for two other cogs, then it stops. When each cog completes its LED-flashing repeat loop it stops and goes into a low-power dormant state.

The "@" prefix on a variable name, such as @StackA, gives software the *memory address* for that variable and not the value of the variable. The cog needs the *address* — not the *name* — of the stack. If you want to know *where* code has saved a variable, use a statement such as *xyz := @testAB* to get the *address*. If you want the *value* saved in testAB, use just the *name*: xyz := testAB.

```
'Program: LED_MultiFlash.spin
VAR   'Variable-definition area
  long  StackA[32]          'Stack workspace for each cog.
  long  StackB[32]
  long  StackC[32]

PUB Main
'Start a cog to flash LED on pin 16, start a 2nd cog to
'flash LED on pin 18, and a 3rd cog to flash LED on pin 20.

   cognew(LED_Flash(16, 30, 10), @StackA)
   cognew(LED_Flash(18, 15, 30), @StackB)
   cognew(LED_Flash(20, 7, 60), @StackC)

'———-Main program ends here———-

'PUB object "LED_Flash" runs run on each cog. Flash LED on Pin 'for
Duration a total of Count times. Duration = 0.01-sec.

PUB LED_Flash(Pin, Duration, Count)
   Duration := clkfreq / 100 * Duration      'Calc cycle duration.
   dira[16..23] := %11111111      'Set pins to output 'mode for LEDs.
   repeat Count             'Loop Count
    !outa[Pin]   'Toggle I/O pin
    waitcnt(Duration + cnt)         'Delay
```

Figure 3: The program listing for the LED_MultiFlash program.

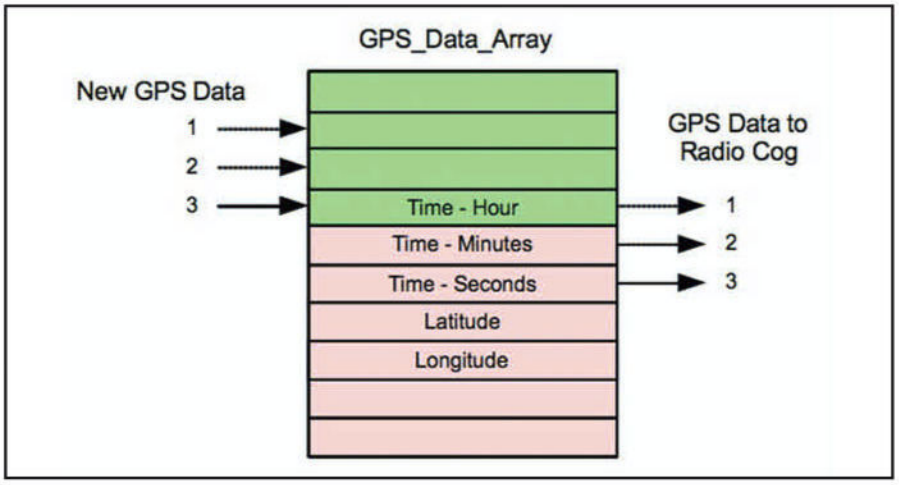


Figure 4: As new GPS values go into the GPS_Data_Array, the radio cog might read a mix of new (green) and old (red) data for a status message. Without some way to control access to an array, cogs can cause memory conflicts.

Photo B. A railway semaphore signal may have one of two conditions-up (horizontal) or down. Some semaphores include colored lenses and a backlit lamp for nighttime signaling. (Courtesy of Wikimedia Commons)

To see how the Propeller uses memory for the LED_MultiFlash program, run it again. Then go to the Propeller Tool's Run tab and click on it. Next choose Compile Current and then choose View Info. You will see a memory map that includes a gray area of initialization code, a red area for the program's code, a yellow area that represents 96 longs of stack memory, and a large purple section of unused RAM.

Four Statements Control Cogs

Before anyone creates a useful Spin program with several cogs, they must understand the four Spin commands that start or stop a cog, identify a cog, and initiate code in a specific cog; *cognew, cogstop, cogid, coginit.*

1. The **cognew** statement requires a longer explanation than the other three, so I'll describe it first. This statement must include the "name" of the object to run in the cog, and a second "@name" that points to a section of memory the cog can use as its stack. The object_name section may include parameters — basically values — to pass to the cog. The following statements show the general format and a specific statement:

```
cognew (object_name (optional_parameters), @stack_name)

cognew(Fluid_Level(10, 30), @FL_Stack)
```


EXPERT LINEARS America LLC

Authorized Distributor for SPE Linear Amplifiers

ExpertLinears.com
281-259-7877

Fully Automatic Solid State Amplifiers!

EXPERT 1K-FA ★ EXPERT 2K-FA
EXPERT 1.3K-FA

THE MOST TECHNOLOGICALLY ADVANCED IN THE WORLD!

Includes Power supply & Fully Automatic Antenna Tuner – Fully Remoteable

Expert Linears America, LLC Offers
SALES & EXPERT SERVICE

★ SPE Factory-trained with 3½ years repairing Expert Amps/Linears
★ Fast Turnaround
★ Over 50 years RF Experience
★ Over 60 years ham radio repair experience

See Hamfest EVENTS ON OUR WEBSITE

EXPERT LINEARS AMERICA, LLC
PO Box 1224
Magnolia, TX 77355
Contact: Bob Hardie W5UQ
Bob@ExpertLinears.com

SOTABeams Now Offers Heavy Duty Wire

Building on the success of its lightweight military green wire, SOTABeams has added a new, heavier duty, antenna wire. The wire is ideal for permanent installation and is rated at 1,000 watts. It is supplied on convenient 50-meter (164-foot) reels, which is enough for most antenna applications and keeps the shipping costs low.

This heavy duty wire has a stranded tinned copper conductor and a UV-stable PVC jacket. The 50-meter reel size is perfect for making most HF antennas, such as an 80-meter dipole. SOTABeams' exclusive military green cable jacket color is unobtrusive in many applications and can be used for antennas, radials and counterpoises. The wire has a resistance of less than 1 Ohm/50 meters.

SOTABeams' new heavy-duty military green wire is available now and is $11.78 per 50-meter spool (add tax if you're in Europe). For more information, contact: SOTABeams 2nd Floor, Paradise Mill, Park Lane, Macclesfield SK11 6TL United Kingdom. Phone: +44 (0) 7976 688359. Website: <http://www.sotabeams.co.uk>.

Specifications:

- 1,000 watts RF rated (50 Ohm)
- Conforms to Military Defense Standard 61-12 pt. 6
- Tinned annealed copper conductor
- 1 mm total conductor CSA
- 32 strands x 0.2-mm copper conductor stranding
- 2.55 mm cable diameter
- Voltage Rating: 1.5-kV rms
- PVC insulation
- Weight 750 g/50 m (approx. 15 g/meter),
- Military Green color jacket
- Current Rating: 10 amps at 25° C
- Resistance: less than 1 ohm per 50 meters

Note: *"What's New" is not a product review and does not constitute a product endorsement by* CQ. *Information is primarily provided by manufacturers/vendors and has not necessarily been independently verified.*

The second statement will start a cog, load an object named Fluid_Level into it and send the Fluid_Level code two parameters, 10 and 30. The overarching Main program must first create an array, FL_Stack. For good measure, you could set aside 50 longs for this stack.

```
VAR
  long  FL_Stack[50]
```

You may specify a larger or smaller stack as needed, and give it a different name, perhaps FluidStack, Sensor_Stack, or something else. You can run a test program to determine stack-space requirements[2].

Although it rarely happens, sometimes a complicated program might use all eight cogs. Then your code needs to start another one. But the Propeller has no free cogs and you don't know it. A cognew statement can include a way to determine whether or not it has taken control of an unused cog. The next statement returns the identification number (ID) of the newly started cog, so you can capture the ID this way:

```
FluidCog := cognew(Fluid_Level(10, 30), @FL_Stack)
```

If FluidCog equals -1, the Propeller chip has no cogs available. Here's a simple way to test for an available cog and remain in a repeat loop until one becomes available:

```
repeat until (cognew(Fluid_Level(10, 30), @FL_Stack) > -1
```

This type of statement also can assign the cog ID number to a variable when your code needs to use it later:

repeat until (**CogNumb :=** cognew(Fluid_Level(10, 30), @FL_Stack)) > -1

2. The **cogstop** statement includes the ID of the cog to stop: cogstop(4) or cogstop(CogNumber), for example. When a cog stops, it goes into a low-power dormant state.

3. You use the **cogid** command to get the number of the cog in which you have included this command. You might need to display that number when debugging code, or use it with a cogstop command to stop execution in the current cog:

cogstop(cogid), or This_cogs_number := cogid.

4. The **coginit** statement operates like a cognew statement except it launches code in a specific cog and it does not return a cog value. (When you start code in a specific cog, you already know its ID.) A complete statement looks like this for code to run in Cog 5:

coginit(**5**, Fluid_Level(10, 30), @FL_Stack)

Avoid Resource Conflicts

Software for the three LED-flashing cogs ran independently and the cogs didn't need to communicate with each other. But suppose we have a device that includes a GPS receiver, environmental sensors, and a radio module that reports status and receives commands. A programmer could assign a cog to each device, and in this case, the cogs must communicate. A cog that handles a humidity sensor, for example, would write a value in a Main-memory RAM location that the cog for the radio could read and then transmit. Although all cogs can access Main memory, only one cog may access it at a time. Thus we call a Main memory location a *mutually exclusive resource.*

The Elecraft K-Line

Now Featuring the New K3S Transceiver

K3S
Superhet/SDR Architecture
Ultra Low-Noise RX/TX

The Elecraft K3 set the standard for compact, high-performance transceivers, proving to be ideal for DXpeditions, multi-transmitter contesting, Field Day, and home stations alike. With the 2nd-generation K3S, we've raised the bar once again, upgrading nearly every subsystem. Improvements include:

- Ultra low-noise synthesizer
- USB port with integrated control and audio
- Second preamp for 12-6 m weak-signal work
- 5/10/15 dB attenuator settings
- ATU option with true bypass relay
- Accurate, high-speed CW even in SPLIT mode
- 100-500 kHz coverage
- Enhanced look and feel; soft-touch VFO knob

P3 Panadapter
Now with TX Signal Monitoring
Out-Performs Built-In Band Scopes

The P3 panadapter's real-time spectral and waterfall displays add a visual dimension to DXing, revealing weak signals you might otherwise miss. The P3 is fully integrated with the transceiver, allowing instant QSY to any signal. Optional TX metering adds power/SWR graphs and signal envelope monitoring.

NEW! SP3 Matching Speaker

(sold separately)

KPA500
Works with Any Transceiver
Silent, Ultra-Fast T/R Switching

The KPA500 amp features instant RF-based band switching, plus remote band selection that tracks the band of the K3S or K3. It has bright alphanumeric status display and LED bar graphs, and a rugged, internal linear supply. The compact KAT500 ATU (not shown) uses a fast, accurate tuning algorithm. Saved matching network settings can be recalled automatically as you tune the transceiver's VFO, so you'll be ready wherever DX appears.

ELECRAFT®

PO Box 69, Aptos, CA 95001 | 831.763.4211 | info@elecraft.com | www.elecraft.com

Exclusive memory access seems like a useful data-protection technique, but it applies to only *one address at a time*. When cogs share more than one memory location, we could have a problem. Consider a cog for the GPS receiver. It saves time, date, longitude, and latitude data in a Main-memory array, GPS_Data_Array. When the Hub gives the radio cog its turn, it takes time information from this array for a status report. The GPS cog has started to put information in the array and just updated the hours value. Now the radio cog takes the *new* hours value and the *old* minutes and seconds values. Thus the status report might give us incorrect values due to overlapping use of a Main-memory array by two cogs (*Figure 4*).

To overcome conflicting uses of an array or other resource you could define a global variable such as array_flag, which all cogs could read or write. The Main program sets the flag to "array-open," to start. The GPS cog would detect the "array-open" flag and immediately change it to the "array-busy" condition. Then the GPS cog would update the time data, after which it would change the flag to "array-open." Any other process that wants to access the GPS array must wait until the GPS update ends.

Code in the radio cog would also examine the array_flag. The "array-busy" state indicates another process has control of the array, so the radio cog must wait its turn. When the flag indicates "array-open," the radio cog changes the flag to 'array-busy" and gathers the GPS information for its report. When the radio cog finishes using the array, it changes the flag back to "array-open."

This type of cooperative use of a resource, here the GPS_Data_Array, can prevent contention for a resource and unexpected and difficult-to-debug conditions. But this technique can fail because the Propeller cannot test a variable — the flag — and change it to True or False in one Hub clock cycle. First, the code must test the flag in Main memory and then wait until its next clock cycle to change the flag's state. During the time between the two clock cycles, a process in another cog could change the flag's state. We need *atomic* flag operations, which means a read *and* a write operation occur during the same period a cog has exclusive access to Main memory.

Lock MCU Resources

To overcome the flag problem, the Propeller Hub gives programmers eight 1-bit "locks" created specifically to help software guard resources. The Hub does not assign a lock to a range of memory addresses or to specific I/O pins. It serves only as a 1-bit flag. Four Spin statements make it easy to work with locks because their operation occurs in a single cog time slot. Statements in other cogs cannot interfere with an atomic lock-bit read-write step. Keep in mind protected "resources" can include more than memory; for example a serial port, access to a sensor, inputs from a touch screen, and so on. Programmers often call this type of 1-bit indicator a *semaphore*, named after a type of railway signal (*Photo B*). As you read the descriptions below, think of a door with a lock you may open or close.

1. A **locknew** statement lets a program request a 1-bit lock from the Hub. The locknew operation returns the lock identification number (0-7) automatically assigned by the Hub:

```
LockID := locknew
```

Software that requests a lock must make the lock ID number available so other cogs know what lock to test and set or

WRO on CD

Back issues of *WorldRadio Online* are now available in PDF format!
Five years of information-packed issue are now available in easy-to-access PDF format for you to read and refer to again and again . . . at a fraction of their original cost!

WRO 2009 Order No.WRO2009 **$12.95**
WRO 2010 Order No.WRO2010 **$12.95**
WRO 2011 Order No.WRO2011 **$14.95**
WRO 2012 Order No.WRO2012 **$14.95**
WRO 2013 Order No.WRO2013 **$24.95**

2009 to 2013
All FIVE years!
Get all five years! All sixty information-packed issues of *WorldRadio Online* - from 2009 to 2013 - to enjoy again and again! All five years for only
Order No.WRO5 **$59.95**

Shipping & Handling: USA - $7 for the first item, $3.50 for second and $2 for each additional item. **Foreign -** calculated by weight & destination and added to your total. **FREE** s&h on orders with merchandise total of $100 or more.

CQ Communications, Inc.
17 West John Street
Hicksville, New York 11801
http://store.cq-amateur-radio.com

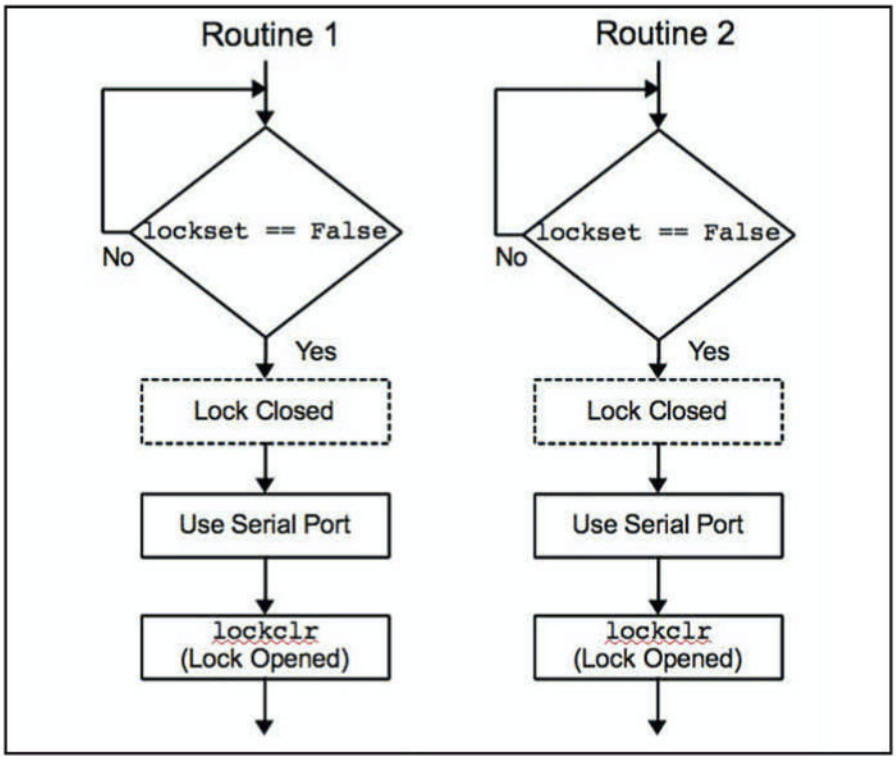


*Figure 5. Flowchart for main operations in program **simple_multicore_demo3d**. The "Lock Closed" blocks (dashed lines) indicate a step performed by the lockset test in the repeat loop. Review the complete program listing to see how the code operates the lock bit.*

clear as they share a given resource. I recommend programmers include the locknew operation in an "if" statement that checks for an available lock before the remaining code runs. Do not assume a locknew request always returns a lock ID number. Other processes might have used all available locks.

```
if(LockID := locknew) == -1        'Get a lock, if available
   pst.Str(string("No lock available."))
else
   lockset(LockID)          'Close lock
   pst.Str(string("Lock checked out: "))
   pst.Dec(LockID)          'Display lock number
   pst.NewLine
```

As soon as this section of code gets a lock assignment (0-7), it closes its assigned lock to block access to the protected resource, and reports the number of the assigned lock.

2. The **lockset(LockID)** operation sets the identified lock (locks the door) to block access to the shared resource by other processes: lockset(LockID). If the operation successfully sets (closes) the lock, it returns a False. If the statement could not set the lock, it returns a True. The True state indicates another process has already set the lock, thus blocking access to the resource.

Before a process tries to use a shared resource it should wait until it gets a False (unlocked) condition returned from the statement lockset(LockID) as shown in the one-statement loop:

```
repeat until lockset(LockID) == False      'Wait for lock to open
```

A True result indicates another process already set (locked) this lock, so the loop waits for a False result. The False condition will let us know the loop's lockset(LockID) step successfully set (closed) the lock, so the code that follows can run.

3. When a cog or process no longer needs to use the shared resource, it sets the lock in the False (unlocked) state with the **lockclr(LockID)** command. This operation returns the previous lock condition, if you need it.

Study with the best!
License Study Materials by
Gordon West, WB6NOA & the W5YI Group
HAM IT UP! A NATIONAL RESOURCE

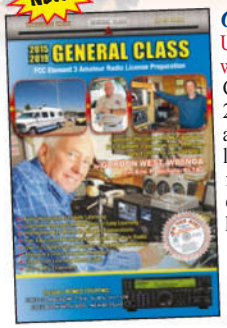
TECHNICIAN CLASS

Technician Class book for the 2014-2018 entry level exam! Gordo reorganizes the Q&A into logical topic groups for easy learning! Key words are highlighted in his explanations to help you understand the material for test success. Web addresses for more than 125 helpful, educational sites. Includes *On The Air* CD demonstrating Tech privileges.
GWTM-14 $21.95

Tech Book & Software Package
Gordo's book with W5YI software for Windows allows you to study at your computer and take practice exams. Explanations from Gordo's book are on the software to reinforce your learning. Includes free Part 97 Rule Book. NCS-14 $29.95

Tech Audio Course on CD
Welcome to Gordo's classroom! Technician audio theory course recorded by Gordo talks you through the Element 2 question pool. Follows the order of his book. On 4 audio CDs. GWTW-14 $29.95

NEW! GENERAL CLASS



General Class book Upgrade to the HF bands with Gordo & W5YI! Gordo's manual for 2015-2019 reorganizes all the questions into logical topic groups for easier learning. His explanations include highlighted key words to help you remember the material for test success. Companion CD is full of great operating tips!
GWGM-15 $24.95

General Book & Software Package
Study at your computer and take practice exams. Windows software includes explanations from Gordo's book, scores your results and highlights areas that need further study. Includes free Part 97 Rule Book. GUS-15 $34.95

General Audio Course on CD
General Theory Course recorded by Gordo is full of the sounds that bring ham radio to life! He talks you through the Element 3 theory to help you understand the material and get you ready for your upcoming exam. On 4 audio CDs. GWGW-15 $29.95

EXTRA CLASS

Extra Class book Go to the top with Gordo! 2012-2016 book includes all Element 4 Q&A reorganized into logical topic groups. Gordo's fun, educational explanations with highlighted keywords, and great memory tricks for those tough theory questions! Wait 'til you meet "Eli the Ice Man!"
GWEM-12 $24.95

Extra Book & Software Package
Study at your computer and take practice exams as the W5YI Windows software scores your results and highlights areas that need further study. Includes explanations from Gordo's book. Free Part 97 Rule Book. ECS-12 $39.95

Extra Audio Course on CD
Extra Class Theory Course recorded by Gordo talks you through the difficult Element 4 theory to help you understand the material and get you ready for your upgrade to the top. On 6 audio CDs.
GWEW-12 $39.95

W5YI Software on Windows-compatible CD-ROM.

Order today from W5YI: 800-669-9594 or on-line: www.w5yi.org
The W5YI Group P.O. Box 200065 Arlington, TX 76006-0065 When ordering, mention this CQ ad for a free gift. CQ

4. Processes should relinquish control of locks when they no longer need them. A cog or object might protect a serial port for a period, but after that, controlled access becomes unimportant. In this case, the code should "return" the lock to the Hub with the **lockret(LockID)** command. Once returned, the Hub may reassign a lock to another process.

Keep in mind that locks do not physically protect resources from use by portions of a program or by buggy code. The locks simply provide a way programs can cooperatively share resources. Programmers must write code to properly check-out locks, test them before they use resources, set them, clear them, and return them to the Hub for reuse by other code. The Hub assigns locks in sequence from ID number 0 through 7.

The program **simple_multicore_demo3d** demonstrates how locks operate to protect the serial I/O pins that communicate with a host PC when two cogs need to use them[3]. You may run this program that includes a 12-second delay so you can switch to the Parallax Terminal window to see the results appear. Ensure you have the program **Parallax Serial Terminal** in your Propeller Workspace folder. The download for this article includes it. *Figure 5* provides a flowchart for the lock operations.

Cogs Borrow Code

Take a close look at the listing for the simple_multicore_demo3d program. The Multicore_Routine_1 and the Multicore_Routine_2 objects run in two new cogs that include statements to display information on the terminal window, via the Propeller's serial-USB connection:

```
pst.Str(string("Multicore_Routine_1, running in cog "))
```

How can these two cogs use the QuickStart board's serial port to transmit characters to a PC? They do not define a serial-communication object and include no serial-communication software. But at the start of the program you see the object (OBJ) section that includes this statement:

```
OBJ
  pst : "Parallax Serial Terminal"
```

I declared the use of objects in the Parallax Serial Terminal software and assigned them the pst. prefix. Any object or set of objects declared in the main OBJ section makes them *globally* available to all software. That means any of the software within the program — even code loaded and run in cogs — may use global objects. We do not have to recreate serial code for each cog that needs it.

If we create a file of LCD-control objects that clear a display, position a cursor, display characters, flash characters, change fonts, and so on, the cogs may use them. That's a good feature because people have already posted sets of LCD objects in the Object Exchange. Now you could dedicate a cog to control the LCD and use objects you know work. This capability avoids duplication of objects and code in each cog, which saves time and effort. And it means any cog can use any object declared in the OBJ section.

Notes:

1. *Methods and Cogs*, Parallax, Inc., 2007. <http://bit.ly/1jz5Iq1>
2. Search the Object Exchange (obex.parallax.com) with keywords "stack length" for a demonstration that shows how to determine stack use for an object. Created by Jeff Martin, 2013.
3. *AN011 Simple Multicore Template for the Propeller P8X32A* Parallax, Inc. <http://bit.ly/1NpkcAs>

We continue KZ1G's tutorial on microcontrollers with a look this time at the math that makes everything happen.

An Introduction to Microcontrollers, Part VI

MCU Math takes Several Routes

BY JONATHAN A. TITUS,* KZ1G

Engineers, experimenters, and hams rely on software tools to simplify writing microcontroller programs. These tools include a compiler or an interpreter that separates us from the complexity of machine-language codes that control an MCU's hardware. A compiler would take the instructions that flash an LED every second...

```
DIRA[24] := 1
repeat
  delay.one_second(1)
  OUTA[24] := 1
  delay.one_second(1)
  OUTA[24] := 0
```

and convert them to the proper codes for the MCU. These codes correspond directly to the 1s and 0s that control MCU hardware. In this case, the compiler would link the code with the delay objects defined earlier in the program. Without a high-level language such as Spin, C, C++, or Python, we must use MCU assembly-language code and create a program such as this:

```
DAT
org 0
Toggle rdlong Delay, #0
      shr Delay, #2
      mov Time, cnt
      add Time, Delay
and so on...
```

That type of programming doesn't look like fun. Imagine the complexity of math operations if we had to write them in an assembly language. Thankfully we avoid that burden and use statements such as: beta := 3 + time, or cost := quantity * price. Math operations use three types of values: *integer, fixed point*, and *floating point*. I'll explain each type briefly and give examples of how they work. Remember, MCUs save information as binary numbers in *bytes* (8 bits), *words* (16 bits), and *longs* (32-bit). The MCU doesn't care what the bits represent. It's up to us and our software to sort that out.

** 14253 S. Trailview Way*
Herriman, UT 84096-8029
email: jontitus@comcast.net

Table 1. Memory allocations for integers by data type

Data Type	Bits	Data Format	
		Unsigned	Signed
Byte	8	0 to 255	-128 to +127
Word	16	0 to 65,535	-32,768 to +32,767
Long	32	0 to 4.29×10^9	-2.15×10^9 to $+2.15 \times 10^9$

Integer Math

Integer math involves only values such as -96, 0, 54, 12, and so on. *Table 1* shows the ranges for signed and unsigned integers saved as bytes, words, or longs. Integers cannot include fractional values such as -90.45 or 854.3. We typically use integers to count events, identify data in arrays, control I/O ports, and for codes that define displayed characters.

You could use an integer to represent a degree position of, say 115°, for an antenna rotor, but you couldn't use an integer if you needed fractional degrees. Thus we have math operations that only use integers and from which we expect integer results. You could multiply 37,684 by 145, but you could not get an accurate square root of 78 with integer math. If you wonder about what happens when an MCU handles integer math, look at Freescale Semiconductor Application Note AN1219 that includes flow charts and assembly-language code[1].

You might remember the use of 2's-complement values to represent signed values as introduced in Part 4 of this series[2]. Here's a quick review: A 2's-complement number uses the most-significant bit as a sign bit. A 0 indicates a positive number and a 1 indicates a negative number. But we can't simply convert 5 (0000001012) to 1000001012 to get -5. First, we complement each bit in 000001012 to get 111110102. Second, we add 1 to the complemented value: 11111010 + 00000001 = 11111011[3]. To prove you have -5 in a signed 8-bit value, add the binary value for 5 and -5:

```
  00000101 =  5
+11111011 = -5
 100000000 =  0
```

Although this addition produces a carry bit, a byte-size result cannot hold it and we ignore it. The same holds true for 16- and 32-bit integer math. Integer operations include addition, subtraction, multiplication, and division, although a division result would not include fractional information or a remainder. Programmers use the 2's-complement method to "create" nega-

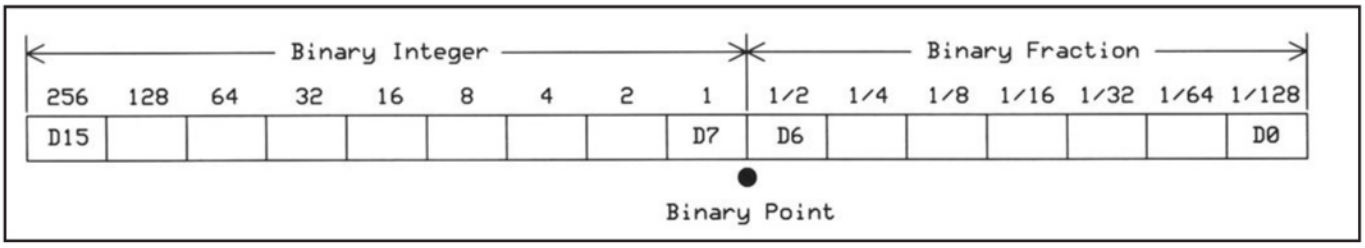


Figure 1. In this word value, I placed a binary point (decimal point) between bits D7 and D6. Bits to the right of the point will represent fractions in my software. It's up to programmers to keep track of a binary point.

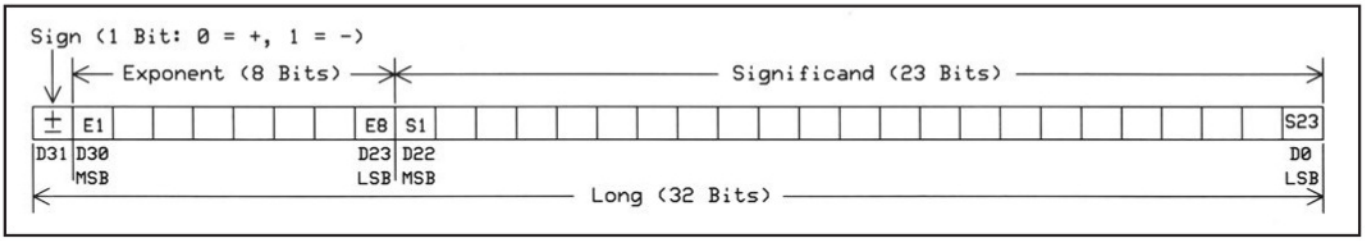


Figure 2. A single-format floating point (FP) value provides 23 bits for a value's significand (D22-D0) and one bit for its sign (D31). The binary exponent comprises eight bits (D30-D23).

tive numbers as needed. The compilers and interpreters handle the signs properly.

You must define variables to hold math results so they have enough bits to provide accurate results. When you multiply the largest unsigned 8-bit value (255 = 111111112) by itself the answer comes to 65,025, or 1111111000000012, which requires 16 bits. When a program uses integer math, ensure you define variables with sufficient bits to hold results.

Fixed-Point Math Handles Fractions

Every time we shop, we see fixed-point numbers; two digits to the right of the decimal point for tenths and hundredths of a dollar (cents) and many digits for dollars to the left — sometimes too many dollars. Programmers may assign a binary point to a value to separate binary integers from binary fractions. The MCU knows nothing about my binary point. We just use it as a convenient way to handle fractions. As noted earlier, to an MCU, all values are simple 1s and 0s.

So what's a binary fraction? Just as integer bits increase as powers of 2, fractions decrease as powers of 1/2: 1/2, 1/4, 1/8, and so on for a 16-bit value with 9 integer bits and 7 fractional bits (*Figure 1*).

The fractional part of the word in *Figure 1* provides a resolution of 1 part in 128, or 0.00078125. Thus we can use this arrangement to represent values with a resolution of about one hundredth (0.01). The maximum unsigned value comes to 511.99. Signed values in this format have a range from +255.99 to -255.00. Bit D15 serves as the sign bit.

Given 16 bits in a word, you might need more integer bits and fewer fractional bits or vice versa. So move the binary point as needed, or use a binary point in a 32-bit, long variable. Just remember the binary point exists only in the way your software interprets the information. For an MCU, a decimal or binary point has no meaning.

What happens to the fixed binary point when we do math? To keep things simple, multiply two 8-bit fixed-point values, 1010.11012 (10.8125) and 0011.00102 (3.125). A decimal calculator shows 33.789. The binary result comes to:

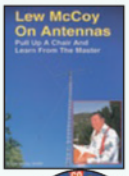
0010_0001_1100_1010

Table 2. Fractional-bit values for floating-point significands

Decimal Value	Bit S	Bit D	Decimal Value	Bit S	Bit D
1	HIDDEN		0.000244140625	S12	D11
0.5	S1	D22	0.0001220703125	S13	D10
0.25	S2	D21	0.00006103515625	S14	D9
0.125	S3	D20	0.000030517578125	S15	D8
0.0625	S4	D19	0.0000152587890625	S16	D7
0.03125	S5	D18	0.00000762939453125	S17	D6
0.015625	S6	D17	0.000003814697265625	S18	D5
0.0078125	S7	D16	0.0000019073486328125	S19	D4
0.00390625	S8	D15	0.00000095367431640625	S20	D3
0.001953125	S9	D14	0.000000476837158203125	S21	D2
0.0009765625	S10	D13	0.000000238418579101562	S22	D1
0.00048828125	S11	D12	0.000000119209289550781	S23	D0

Lew McCoy on Antennas

by Lew McCoy, W1ICP



Unlike many technical publications, Lew presents his invaluable antenna info in a casual, non-intimidating way for anyone!

8.5 X 11 Paperback $19.95
CD Version $14.95
Buy both for only $29.95

The NEW Shortwave Propagation Handbook

by W3ASK, N4XX & K6GKU

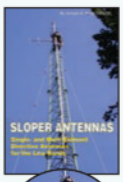
A shortwave propagation book with information on sunspot activity, propagation predictions, unusual propagation effects and do-it-yourself forecasting tips.

8.5 X 11 Paperback $19.95
CD Version $14.95
Buy both for only $29.95

Sloper Antennas

By Juergen A. Weigl, OE5CWL

Single- and Multi-Element Directive Antennas for the Low Bands



With calculations and practical experience, this book shows which basi concepts have to be considered for sloper antennas for the low bands.

6 X 9 Paperback $24.95
CD Version $18.95
Buy both for only $36.95

Shipping & Handling: U.S. add $7 for the first item, $3.50 for the second and $2 for each add'l item. FREE shipping on orders over $100 to one U.S. address. CN/FN add $25 for 1st item, $10 for 2nd and $5 for each additional. Buy Both=single item!

CQ The Radio Amateur's Journal

Phone 516-681-2922
FAX 516-681-2926
http://store.cq-amateur-radio.com

But where did the binary point go in the answer? When we multiply decimal numbers such as 2.13 * 1.7, we get 3621. Then we move the decimal point from the right side of the answer three decimal places to the left to get 3.621. We used four bits for each fraction in a multiplied value, so we move the binary point 4 + 4, or eight positions from the right to give us:

0010_0001.1100_1010

Math operations cannot create additional significant figures, so we must truncate or round the answer. Suppose you need only four fractional bits. A shift of the value four bit positions to the right does the job:

0010_0001.1100_1010 → 0000_0010_0001.1100

But you might want to round the answer to the nearest 1/16th before you shift the answer. The following step adds 1/32 to the multiplication answer. If the answer has the 1/32 bit set, the sum creates a carry bit to round the answer to the nearest 1/16:

0010_0001.1100_1010 = Answer
+0000_0000.0000_1000 = 1/32nd
0010_0001.1101_0000 = Answer rounded to nearest 1/16th

Now shift the answer four bits to the right to give: 0000_0010_0001.1101. Fixed-point math operations can become a chore because we must "follow" the decimal point through each step. And debugging fixed-point operations can consume a lot of time. Software tools for most MCUs include a fixed-point library that simplifies programming[4].

Floating-Point Math

We use floating-point math without giving it a second thought because calculators handle decimal points and exponents for us. And programmers use floating-point (FP) math because it simplifies operations such as square roots for LC-filter calculations and logarithm conversions for decibel ratios. Suppose we want to calculate the center frequency for an LC filter created with a 1000-pF capacitor and a 127-nH inductor. We use the equation below:

$$f = \frac{1}{2\pi\sqrt{LC}}$$

We need values in units of Farads and Henrys: 1,200 pF = 1.2 x 10^{-9} F, and 127 nH = 127 x 10^{-9} H respectively. To get the answer for L times C, we multiply the significands[5]: 1.2 and 127 to get a new significand: 152. Then we add the exponents, -9 and -9, to get -18. The complete answer for L*C equals 152 x 10^{-18} FaradHenrys. After the remaining calculations, the frequency value equals 12.9 x 106 Hz, or 12.9 MHz. In this example, the decimal point "floated" from the left side of 0.000000000000000152 to the right side of the answer 12900000.

In 1985, the Institute of Electrical and Electronics Engineers (IEEE) adopted a standard format to represent floating-point values for computer math. People refer to the standard as IEEE 754-2008 or as IEEE-754[6]. Computer FP operations combine a binary significand and a binary exponent in much the same way we used decimal values above. Thankfully, software and hardware handle the details of FP math for us. The descriptions that follow refer to 32-bit FP math, or single format values. A 64-bit version, the double format, exists, but it's unlikely you'll need it. *Figure 2* shows a single-format floating-point value.

Given the IEEE-754 format, we express a single value as:

$$\text{Value} = Sg2^{E} \quad \text{where } 1_{10} \leq S < 2_{10}$$

In other words, we always have significand values (S) between 1.00000... and 1.9999999... The people who created the IEEE standard realized they could "leave out" the 1 in the significand and add it back later. They called this a "hidden bit," but it doesn't really get hidden. It reappears automatically in the answer, but doesn't appear in FP data saved in memory. Software or hardware automatically handles

```
; ; Calculate resonant frequency
; inductanceH  equ F10       ;Store inductance in reg F10
; capacitanceF equ F11       ;Store capacitance in reg F11
; frequency    equ F12       ;Store freq result in reg F12
;
; inductanceH = 127E-9
   SELECTA, 10                ;Select F10 register
   FWRITEA, 34, 08, 5D, 7F  ;Save this value in it
;
; capacitanceF = 1200E-12
   SELECTA, 11                ;Select F11 register
   FWRITEA, 30, A4, ED, 3F  ;Save this value in it
;
; frequency = 1/(PI * 2 * SQRT(inductanceH * capacitanceF))
;
   SELECTA, 12                ;Select F12 register
   FSETI, 1                   ;Save the value 1 in it
   LEFT                       ;Left parentheses,
   FWRITEA, 40, C9, 0F, DB  ;so save this value in temporary
                              ;storage. It is 2 * PI
   LEFT                       ;Left parentheses
   FSET, 10                 ;Get value in F10
   FMUL, 11                 ;Multiply it by value in F11
   SQRT                     ;Take square-root of the result
   RIGHT                    ;Right parentheses
   FMUL0                    ;Multiply by contents in reg 0,
                            ;which holds (2*PI)
   RIGHT                    ;Right parentheses
   FDIV0                    ;Divide the result into 1,
                            ;(held in reg 12)
                            ;leave answer in reg 12
```

Figure 3. Code for the instructions produced by the compiler so an FPU V3.1 IC can calculate the resonant frequency of an LC filter. I have added comments that explain the operations.

the hidden bit for us. If you simply want to use FP math and don't want details, feel free to skip the next section that "decodes" a 32-bit FP value and explains how the significand and exponent get used.

Values Go from Decimal to FP

Floating-point operations usually need a conversion of decimal values from our world into the proper FP format for an MCU's circuits and vice versa. Thankfully, software or hardware handles this process. But it helps to see what an FP value looks like and what the bits mean. I used an online tool to convert the value -1234510 into an FP value[7]: -12345 becomes C640E400 hex, or in binary notation (underlines added for clarity):

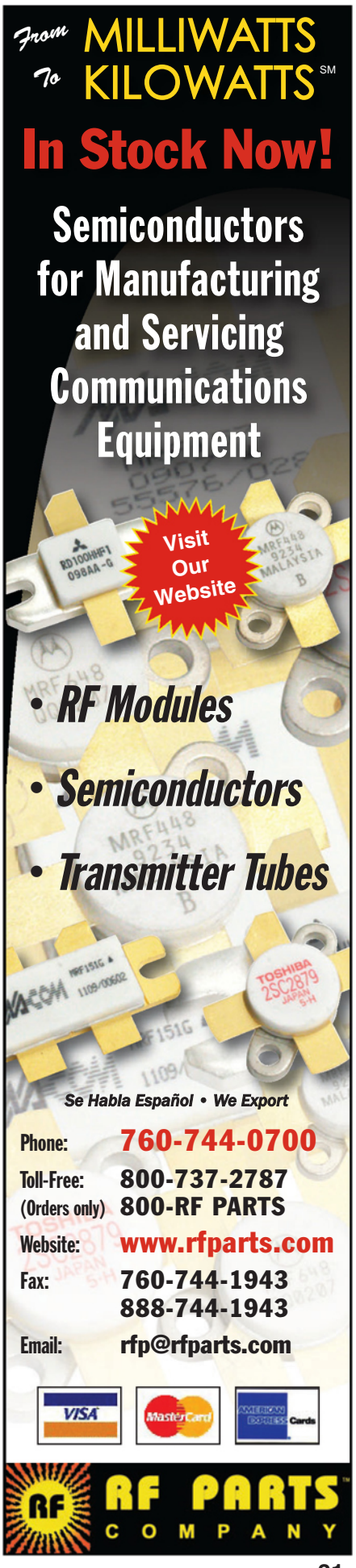
$$1100\_0110\_0100\_0000\_1110\_0100\_0000\_0000_2$$

Based on the arrangement of bits shown in *Figure 2*, the orange bit indicates the sign of the significand: 0 for plus and 1 for minus. In this example, the exponent (E) 100011002 equals 14010 (128 + 8 + 4). At this point, the exponent does not have a sign, plus or minus. The significand (S) in blue supplies the fractional value explained earlier. The binary point for this fraction goes to the extreme left:

.100_0000_1110_0100_0000_0000

To interpret a single-format value, start with the significand. Get the fractional-bit values from *Table 2* for each logic-1 in the fraction and add them. Start with the left-hand bit and continue to the right (S1 + S8 + S9 + S10, and so on...). The example above yields 0.50695800810. Then add the "hidden" bit that always equals 1.000... The final significand amounts to 1.50695800810.

Next we must multiply the significand by 2E, and so far E equals 140. The IEEE-754 standard does not give exponents a sign. Instead it biases the exponent by



127, so we must subtract 127 from the exponent value (140 - 127 = 13). Now we have 213, or 819210. Multiply 1.50695800810 by 819210 and we get 1234510. After we insert the sign identified by bit D31, we have -12345.

About the FP Software

Programmers have created floating-point libraries of math functions that we use often to save time and reduce debugging chores. MCU manufacturers and programming-tool suppliers make these libraries available for free because they want us to buy their MCUs. I have used programs from the Propeller Object Exchange (OBEX), in particular the public object cfloatstr in the clib package created by Dave Hein. Search the OBEX with the keyword "float" to find other floating-point software such as the Floating Point library of math functions and documents. Cam Thompson, an FP expert, created this software. Thanks go to Mike Green for help understanding pointers.

Substitute Hardware for Software

Many engineers frown on adding hardware to a design when they can use software instead. But software consumes memory and processor time, so add-on hardware has its place. If you plan to make extensive use of FP values, consider a hardwired FP "engine" or FP unit (FPU), either as a stand-alone IC or built into an MCU. The ARM Cortex-M4F processor core includes an FPU. Companies such as Texas Instruments, Atmel, Freescale, Silicon Labs, NXP, and STMicroelectronics offer M4F processors. Compilers can take advantage of an FPU and they provide specific FP instructions and operations.

Years ago, I discovered the Micromega ICs created specifically to do FP-math operations. The 32-bit version, uM-FPU V3.1, handles single-format FP values and it communicates via an SPI or an I2C port, which most MCUs have[8]. The FPU V3.1 IC furnishes 128 32-bit FP registers and it has a set of over 200 instructions, although you'll probably use only a small subset. Those codes get sent from the host MCU to the FPU-V3.1 IC that performs the operations and saves results. The host MCU can obtain calculation results from registers in a FPU V3.1 IC.

Don't let the number of instructions discourage you. A free Micromega compiler translates high-level math operations such as X = SQRT(inductanceH * capacitanceF) into the proper instructions. You keep track of FPU-3.1 register, to which you can assign names such as inductanceH and frequency that make it easy to track their use. *Photo A* shows the Micromega compiler's high-level code window with several statements we can immediately understand.

The code in *Figure 3* shows the instructions produced by the compiler so an FPU V3.1 IC can calculate the resonant frequency of an LC filter. I have added comments that explain the operations. Note the compiler automatically converted the decimal numbers to floating-point values, expressed as hexadecimal bytes:

After the compiler does its thing, we must send these commands and values to an FPU V3.1 IC. Once again, we stand on the shoulders of giants who created I2C and FPU code we can adapt to our needs. Several programs help us transfer commands and data to an FPU V3.1 IC. At the end of the calculation, our MCU can get the frequency value in decimal form and display it via the Parallax Serial Terminal, or any terminal. My software and an FPU V3.1 IC calculated 12.89 MHz for the LC filter.

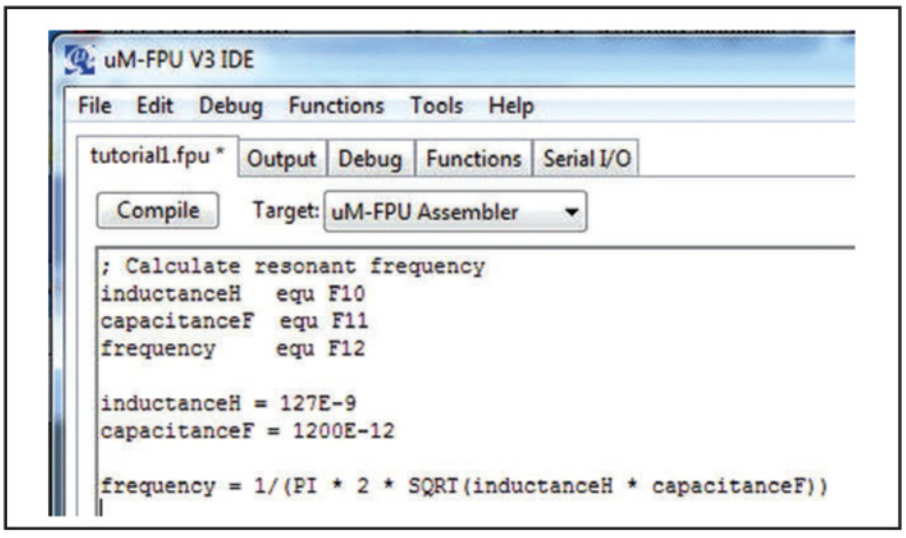

uM-FPU V3 IDE
File Edit Debug Functions Tools Help
tutorial1.fpu * Output Debug Functions Serial I/O
Compile Target: uM-FPU Assembler
; Calculate resonant frequency
inductanceH equ F10
capacitanceF equ F11
frequency equ F12
inductanceH = 127E-9
capacitanceF = 1200E-12
frequency = 1/(PI * 2 * SQRT(inductanceH * capacitanceF))

Photo A. This image shows the high-level source code written to calculate the frequency of an LC filter. I assigned values and the answer to specific 32-bit registers. The compiler creates code for an FPU V3.1 IC.

Wrapping Up…

The next (and final) installment of this series will explain how an MCU communicates with an FPU V3.1 chip, and I'll discuss some debugging techniques and tips.

Notes

1. Johnson, Mark, "M68HC08 Integer Math Routines," AN1219, Freescale Semiconductor. <http://bit.ly/210W5ox>.
2. Titus, "An Introduction to Microcontrollers, Part IV," *CQ* January 2016, p. 32
3. For a 2's complement converter, visit: <http://bit.ly/1RQCodh>.
4. For a set of fixed-point-math objects, see "Fixed-point math with trig functions in SPIN" at: <http://obex.parallax.com/object/232>.
5. The significand represents the value in a number expressed with an exponent. If you have a number such as 78.12, you can represent it as 7812 x 10^{-2} (10 raised to the -2 power). The 7812 is the significand and the -2 is the exponent. So the significand represents the value and the exponent determines where to put the decimal point, or, when working with 1s and 0s, the binary point.
6. Overton, Michael L., "Numerical Computing with IEEE Floating Point Arithmetic," Society of Industrial and Applied Mathematics, Philadelphia, PA, 2001.
7. For an IEEE-754 converter, please visit: <http://bit.ly/1lo7klb>.
8. For Micromega information, please visit: <http://micromegacorp.com/umfpu-v3.html>.